

计算机常用数值算法与程序（C++版）

◆ 何渝 编著

人民邮电出版社
POSTS & TELECOM PRESS

计算机常用数值算法与程序

(C++版)

何 渝 编著

人民邮电出版社

图书在版编目(CIP)数据

计算机常用数值算法与程序(C++版)/何渝编著．北京：人民邮电出版社，2003.7
ISBN 7-115-11183-9

Ⅰ.计… Ⅱ.何… Ⅲ.C语言—程序设计—数值计算—计算方法 Ⅳ.TP312

中国版本图书馆CIP数据核字(2003)第018316号

内 容 提 要

本书主要介绍利用计算机解决科学研究、工程设计、经济与商业管理等众多领域中数值计算所涉及到的数学问题的各类数值计算算法，并利用计算机程序设计语言C++编程实现这些算法。

本书包含复数运算，随机数生成，多项式计算，矩阵运算，插值，数值积分，线性方程组求解，矩阵特征值及特征向量计算，非线性方程(组)求解，常微分方程(组)求解，拟合与逼近，最优化，数学变换，特殊函数等上百个算法。书中的每种算法都在Visual C++ 6.0环境下编译调试通过，如果需要可以将书中的程序移植到其他软硬件计算平台的环境中。

本书可以作为大专、本科及研究生“数值分析”(计算方法)、“工程数学”、“概率与统计”、“数据处理”、“经济数学”等相关课程的教学参考书，也可以作为科研工作者进行科学研究，工程技术人员进行工程设计开发计算分析，管理人员进行科学管理时的技术参考手册，其中所定义的类、方法(函数)可以直接用于计算机软件的开发使用。

计算机常用数值算法与程序（C++版）

◆ 编　　著　何　渝
责任编辑　邹文波

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
读者热线　010-67129260
北京汉魂图文设计有限公司制作
北京隆昌伟业印刷有限公司印刷
新华书店总店北京发行所经销

◆ 开本：787×1092　1/16
印张：31.75
字数：772千字　　2003年7月第1版
印数：1-5 000册　　2003年7月北京第1次印刷

ISBN 7-115-11183-9/TP·3393

定价：49.00元(附光盘)

本书如有印装质量问题，请与本社联系　电话：(010)67129223

前　言

自1946年电子计算机问世以来，数值计算一直是计算机应用的主要领域之一。利用计算机进行数值计算，一方面要有适合于计算机特点的各种有效的计算算法(本书收集整理了多种算法，汇集成册，以利查阅使用)，另一方面为了将数值计算的各种算法得以在计算机上实现，需要借助于计算机的程序设计语言。最早出现的计算机高级程序设计语言—— Fortran，就是专门为了实现数值计算面发明的。随着计算机程序设计的发展，Fortran 逐渐让位给了 C。近年来随着面向对象概念与思想对计算机程序开发的影响，程序设计语言方面的代表C++语言被广泛接受与采用，特别是在企业界与教育科研界。为此，本书采用C++语言编写了数值计算的各种常用算法，以满足各学科与各行业中教学、科研与计算机工程开发人员的需要。本书中的每一个算法都在奔腾 PC 机上用美国 Microsoft(微软)公司出品的 Visual C++ 6.0 版本调试通过的。本书采用的是标准C++，可以很容易地将这些程序移植到其他不同的计算机硬件和软件平台上使用。书中的算法程序以及示例都可用作学习数值算法和C++语言的范例，也可直接或稍加修改应用于实际的软件开发中。

本书第1章专门对一些特别的问题进行了介绍，读者最好先阅读一下该章的内容。这对于理解学习书中后面的内容，特别是对于那些不熟悉数值算法，不知如何使用Visud C++ 6.0，不知如何在Visud C++6.0 环境下对程序进行编辑修改、编译连接、运行调试的读者很有益处。

本书对各数值算法，以及C++语言的使用都有详细的解释，对所引用的标准与用法都进行了说明，以便于读者能够更好地理解与消化。

书后配有光盘，其中包含有本书所有算法的C++源程序代码文件、示例程序代码文件、示例程序的可执行文件和示例程序的输出结果文件，可以作为学习、参考及应用开发软件的文档及资料，并可省去读者输入C++源程序代码之苦。光盘中附有使用说明文件，对于如何操作使用光盘中的内容，作了详细的说明。

在本书的撰写中，龚敏敏、张桀宁参加了本书部分算法C++程序的编写与调试，向华、孙利民、王霞、秦泽辉、高杨、何佳、张海伟、陆明翔参与了本书部分章节文字录入与准备。谨此一并表示深切的谢意。

由于作者水平所限，书中难免有错误及不妥之处，恳请广大读者批评指正。

作者

2003 年 5 月

于北京工商大学计算机学院

目　录

第 1 章　概论 …… 1
1.1　类库 …… 2
1.2　头文件及方法(函数)的命名 …… 2
1.3　头文件的使用 …… 2
1.3.1　头文件 LinearEquation.h …… 3
1.3.2　头文件 LinearEquation.inl 节选 …… 4
1.4　模板 …… 6
1.5　重载 …… 8
1.5.1　重载函数 …… 8
1.5.2　重载运算符 …… 8
1.6　浮点数的比较 …… 8
1.7　公共头文件 …… 9
1.7.1　comm.h …… 10
1.7.2　示例 …… 11
1.8　Visual C++ 6.0 的编译运行环境 …… 14
1.8.1　VC 6.0 工程(文件夹)的创建 …… 14
1.8.2　VC 6.0 工程的设置 …… 18
1.8.3　VC 6.0 程序编制调试 …… 20
1.8.4　注意事项 …… 21
1.9　输出 …… 22
1.9.1　在程序中加特别的输出语句 …… 22
1.9.2　利用命令行命令重定向功能 …… 23
第 2 章　复数运算 …… 24
2.1　复数概念 …… 24
2.1.1　复数定义 …… 24
2.1.2　模与辐角 …… 24
2.1.3　共轭复数 …… 24
2.1.4　虚数单位乘方 …… 24
2.1.5　复数的表示法 …… 25
2.1.6　复数运算 …… 25
2.2　头文件<complex>及模板类 complex …… 26
2.2.1　头文件<complex>描述 …… 26
2.2.2　模板类 complex …… 29
2.3　模板类 complex 与<complex>成员 …… 30

2.3.1 类型定义 …… 30
2.3.2 模板类 complex<float> …… 30
2.3.3 模板类 complex<double> …… 30
2.3.4 模板类 complex<long double> …… 31
2.3.5 构造函数 …… 31
2.3.6 取实数部分 …… 31
2.3.7 取虚数部分 …… 32
2.3.8 复数实数部分 …… 32
2.3.9 复数虚数部分 …… 32
2.3.10 赋值 …… 32
2.3.11 赋值加 …… 32
2.3.12 赋值减 …… 33
2.3.13 赋值乘 …… 33
2.3.14 赋值除 …… 33
2.3.15 加法 …… 34
2.3.16 减法 …… 34
2.3.17 乘法 …… 34
2.3.18 除法 …… 34
2.3.19 复数绝对值 …… 35
2.3.20 复数辐角 …… 35
2.3.21 共轭复数 …… 35
2.3.22 正弦 …… 35
2.3.23 余弦 …… 35
2.3.24 双曲正弦 …… 35
2.3.25 双曲余弦 …… 35
2.3.26 指数 …… 36
2.3.27 自然对数 …… 36
2.3.28 以 10 为底的复数对数 …… 36
2.3.29 平方 …… 36
2.3.30 复数极坐标值 …… 36
2.3.31 乘幂 …… 36
2.3.32 平方根 …… 37
2.3.33 判断两复数相等 …… 37
2.3.34 判断两复数不相等 …… 37
2.3.35 复数输入 …… 37
2.3.36 复数输出 …… 38
2.3.37 宏_STD_COMPLEX …… 38
2.4 复数运算示例 …… 38
2.4.1 示例 …… 38
2.4.2 示例运行结果 …… 39

2.4.3 示例说明 …… 40
第3章 随机数生成 …… 42
3.1 随机数生成与头文件 …… 42
3.1.1 随机数生成 …… 42
3.1.2 random.h …… 42
3.2 随机数生成算法 …… 43
3.2.1 产生一个介于0～32767之间的随机整数 …… 43
3.2.2 产生一个[0,1]区间内均匀分布伪随机数 …… 45
3.2.3 产生多个[0,1]区间内均匀分布伪随机数 …… 46
3.2.4 产生任意[a,b]区间内一个均匀分布伪随机整数 …… 47
3.2.5 产生任意[a,b]区间内均匀分布伪随机整数序列 …… 48
3.2.6 产生一个任意均值与方差的正态分布随机数 …… 50
3.2.7 产生任意均值与方差的正态分布随机数序列 …… 51
第4章 多项式与连分式计算 …… 53
4.1 多项式与头文件 …… 53
4.1.1 基本概念 …… 53
4.1.2 polynomials.h …… 54
4.2 多项式与连分式算法及程序 …… 55
4.2.1 计算一维多项式值 …… 55
4.2.2 计算一维多项式组值 …… 58
4.2.3 计算二维多项式值 …… 61
4.2.4 计算一维多项式相乘 …… 65
4.2.5 计算一维多项式相除 …… 68
4.2.6 计算连分式值 …… 71
第5章 矩阵运算 …… 73
5.1 矩阵概念 …… 73
5.1.1 矩阵 …… 73
5.1.2 矩阵基本运算 …… 74
5.1.3 几类特殊矩阵 …… 74
5.2 模板类matrix及头文件 …… 76
5.2.1 模板类matrix与矩阵运算 …… 76
5.2.2 matrix.h …… 76
5.2.3 矩阵模板类matrix说明 …… 85
5.2.4 矩阵基本运算示例 …… 91
5.3 矩阵转置与矩阵判别 …… 96
5.3.1 矩阵转置 …… 96
5.3.2 对称矩阵 …… 98
5.3.3 正定(对称)矩阵 …… 100
5.4 矩阵行列式及矩阵的秩 …… 103
5.4.1 矩阵行列式 …… 103

5.4.2 矩阵秩 …… 105
5.5 逆矩阵 …… 107
5.5.1 高斯—约当法求逆矩阵 …… 107
5.5.2 对称正定矩阵的逆矩阵 …… 111
5.5.3 托伯利兹矩阵的逆矩阵 …… 113
5.6 矩阵分解 …… 116
5.6.1 实矩阵的三角(*LU*)分解 …… 116
5.6.2 一般实矩阵的 *QR* 分解 …… 119
5.6.3 对称正定矩阵的乔里斯基分解及行列式值 …… 122
5.6.4 一般实矩阵的奇异值分解 …… 125
5.6.5 广义逆的奇异值分解 …… 129
第 6 章 矩阵特征值与特征向量的计算 …… 132
6.1 矩阵特征值、特征向量度与头文件 …… 132
6.1.1 基本概念 …… 132
6.1.2 头文件 …… 133
6.2 矩阵特征值与特征向量计算 …… 134
6.2.1 约化对称阵为对称三对角阵的豪斯荷尔德变换法 …… 134
6.2.2 实对称三角阵全部特征值及特征向量 *QR* 法 …… 137
6.2.3 约化一般实矩阵为赫申伯格阵的初等相似变换法 …… 140
6.2.4 求赫申伯格阵全部特征值 *QR* 法 …… 142
6.2.5 实对称阵特征值及特征向量雅可比法 …… 145
6.2.6 实对称阵特征值及特征向量雅可比过关法 …… 148
第 7 章 线性方程组求解 …… 151
7.1 线性方程组与头文件 …… 151
7.1.1 基本概念 …… 151
7.1.2 Interpolation.h …… 152
7.2 求解线性方程组 …… 154
7.2.1 全选主元高斯消元法 …… 154
7.2.2 全选主元高斯—约当消元法 …… 157
7.2.3 三对角方程组的追赶法 …… 161
7.2.4 一般带型方程组求解 …… 163
7.2.5 对称方程组的分解法 …… 167
7.2.6 对称正定方程组的平方根法 …… 170
7.2.7 大型稀疏方程组全选主元高斯—约当法 …… 173
7.2.8 托伯利兹方程组的列文逊法 …… 174
7.2.9 高斯—赛德尔迭代法 …… 178
7.2.10 对称正定方程组的共轭梯度法 …… 180
7.2.11 线性最小二乘问题的豪斯荷尔德变换法 …… 182
7.2.12 线性最小二乘问题的广义逆法 …… 185
7.2.13 病态方程组求解 …… 188

第 8 章　非线性方程(组)求解 …… 190
8.1　非线性方程和非线性方程组与头文件 …… 190
8.1.1　基本概念 …… 190
8.1.2　NonLinearEquation.h …… 191
8.2　各种非线性方程(组)计算方法 …… 193
8.2.1　二分法 …… 193
8.2.2　实根牛顿法 …… 195
8.2.3　实根埃特金迭代法 …… 197
8.2.4　实根连分式法 …… 199
8.2.5　实系数代数方程全部根的 *QR* 法 …… 201
8.2.6　代数方程全部根的牛顿一下山法 …… 203
8.2.7　非线性方程组一组实根的梯度法 …… 206
8.2.8　非线性方程组一组实根的拟牛顿法 …… 209
8.2.9　非线性方程组最小二乘解的广义逆法 …… 213
8.2.10　非线性方程一个实根的蒙特卡洛法 …… 217
8.2.11　非线性方程一个复根的蒙特卡洛法 …… 219
8.2.12　非线性方程组一组实根的蒙特卡洛法 …… 222
第 9 章　插值 …… 226
9.1　插值与头文件 …… 226
9.1.1　基本概念 …… 226
9.1.2　Interpolation.h …… 227
9.2　各种插值计算方法 …… 229
9.2.1　一元全区间不等距插值 …… 229
9.2.2　一元全区间等距插值 …… 231
9.2.3　一元三点不等距插值 …… 233
9.2.4　一元三点等距插值 …… 234
9.2.5　连分式不等距插值 …… 236
9.2.6　连分式等距插值 …… 238
9.2.7　埃尔米特不等距插值 …… 239
9.2.8　埃尔米特等距插值 …… 241
9.2.9　埃特金不等距逐步插值 …… 243
9.2.10　埃特金等距逐步插值 …… 245
9.2.11　光滑不等距插值 …… 246
9.2.12　光滑等距插值 …… 249
9.2.13　第一种边界条件的三次样条函数插值、微商与积分 …… 251
9.2.14　第二种边界条件的三次样条函数插值、微商与积分 …… 255
9.2.15　第三种边界条件的三次样条函数插值、微商与积分 …… 258
9.2.16　二元三点插值 …… 262
9.2.17　二元全区间插值 …… 264

第 10 章　数值积分 …… 267
10.1　数值积分与头文件 …… 267
10.1.1　基本概念 …… 267
10.1.2　Integral.h …… 268
10.2　各种数值积分计算方法 …… 269
10.2.1　变步长梯形法求积 …… 269
10.2.2　变步长辛卜生法求积 …… 271
10.2.3　自适应梯形法求积 …… 273
10.2.4　龙贝格法求积 …… 275
10.2.5　一维连分式法求积 …… 277
10.2.6　高振荡函数法求积 …… 279
10.2.7　勒让德一高斯法求积 …… 281
10.2.8　拉盖尔一高斯法求积 …… 284
10.2.9　埃尔米特一高斯法求积 …… 286
10.2.10　切比雪夫法求积 …… 287
10.2.11　蒙特卡洛法求积 …… 289
10.2.12　二重变步长辛卜生法求积 …… 291
10.2.13　多重高斯法求积 …… 293
10.2.14　二重连分式法求积 …… 296
10.2.15　多重蒙特卡洛法求积 …… 298
第 11 章　常微分方程(组)求解 …… 301
11.1　常微分方程与头文件 …… 301
11.1.1　基本概念 …… 301
11.1.2　OrdinaryDifferentialEguation.h …… 302
11.2　常微分方程(组)计算方法 …… 304
11.2.1　定步长全区间积分的欧拉法 …… 304
11.2.2　变步长积分欧拉法 …… 307
11.2.3　全区间定步长积分维梯法 …… 310
11.2.4　全区间定步长积分龙格一库塔法 …… 313
11.2.5　变步长积分龙格一库塔法 …… 316
11.2.6　变步长积分一步基尔法 …… 318
11.2.7　全区间变步长积分基尔法 …… 321
11.2.8　变步长积分默森法 …… 324
11.2.9　积分一步连分式法 …… 327
11.2.10　全区间积分连分式法 …… 330
11.2.11　全区间积分双边法 …… 333
11.2.12　全区间积分阿当姆斯预报一校正法 …… 336
11.2.13　全区间积分哈明法 …… 339
11.2.14　积分一步特雷纳法 …… 342
11.2.15　全区间积分特雷纳法 …… 346

11.2.16 积分刚性方程组吉尔法 …… 348
11.2.17 二阶微分方程边值问题数值解法 …… 359
第 12 章 数值拟合与逼近 …… 363
12.1 基本概念与头文件 …… 363
12.1.1 数值拟合与逼近的基本概念 …… 363
12.1.2 FittingApproximation.h …… 363
12.2 数值拟合与逼近计算方法 …… 364
12.2.1 最小二乘曲线拟合 …… 364
12.2.2 切比雪夫曲线拟合 …… 368
12.2.3 最佳一致逼近多项式里米兹法 …… 370
12.2.4 矩形域的最小二乘曲面拟合 …… 372
第 13 章 数据处理与回归分析 …… 377
13.1 基本概念与头文件 …… 377
13.1.1 数据处理与回归分析的基本概念 …… 377
13.1.2 Statistic.h …… 377
13.2 数据处理与回归分析计算方法 …… 378
13.2.1 随机样本分析 …… 378
13.2.2 一元线性回归分析 …… 381
13.2.3 *n* 元线性回归分析 …… 383
13.2.4 逐步回归分析 …… 387
13.2.5 半对数数据相关 …… 396
13.2.6 对数数据相关 …… 398
第 14 章 极值 …… 401
14.1 极值与头文件 …… 401
14.1.1 基本概念 …… 401
14.1.2 Extremum.h …… 401
14.2 极值数值计算方法 …… 403
14.2.1 一维极值连分式法 …… 403
14.2.2 *n* 维极值连分式法 …… 405
14.2.3 线性规划 …… 408
14.2.4 *n* 维极值单形调优法 …… 412
14.2.5 *n* 维极值复形调优法 …… 415
14.2.6 确定一元函数极小值点所在区间 …… 420
14.2.7 一元极小值黄金分割法 …… 422
14.2.8 一元不用导数极小值布伦特法 …… 426
第 15 章 数学变换与滤波 …… 432
15.1 数学变换与滤波的说明与头文件 …… 432
15.1.1 数学变换与滤波的说明 …… 432
15.1.2 Transform.h …… 432
15.2 数学变换与滤波数值计算方法 …… 433

15.2.1 傅里叶级数逼近 …… 433
15.2.2 快速傅里叶变换 …… 437
15.2.3 快速沃什变换 …… 442
15.2.4 五点三次平滑(曲线拟合) …… 444
15.2.5 离散随机线性系统的卡尔曼滤波 …… 446
15.2.6 $\alpha-\beta-\gamma$ 滤波 …… 451
第 16 章 特殊函数 …… 456
16.1 特殊函数与头文件 …… 456
16.1.1 特殊函数 …… 456
16.1.2 SpecialFunction.h …… 457
16.2 特殊函数计算方法 …… 459
16.2.1 伽马函数 …… 459
16.2.2 不完全伽马函数 …… 461
16.2.3 误差函数 …… 463
16.2.4 第一类整数阶贝塞尔函数 …… 464
16.2.5 第二类整数阶贝塞尔函数 …… 468
16.2.6 变型第一类整数阶贝塞尔函数 …… 471
16.2.7 变型第二类整数阶贝塞尔函数 …… 473
16.2.8 不完全贝塔函数 …… 476
16.2.9 正态分布函数 …… 479
16.2.10 t-分布函数 …… 480
16.2.11 x^2-分布函数 …… 482
16.2.12 F-分布函数 …… 483
16.2.13 正弦积分 …… 485
16.2.14 余弦积分 …… 486
16.2.15 指数积分 …… 488
16.2.16 第一类椭圆积分 …… 489
16.2.17 第二类椭圆积分 …… 491

第1章 概 论

当年人类发明电子计算机，主要是进行科学计算。今天，计算机科学与技术得到了极大的发展，电子计算机的应用已经深入、普及到了人类生产与生活的各个方面、各个领域，“计算”的概念早已不是当初发明计算机的科学家和工程师们的概念，计算机的应用也不仅仅限于当初的军事领域，它的概念与应用都大大地扩展了，远远超出了当初人们的想象。但是，当今计算机的主要应用之一仍然是科学计算，也即解决大量的存在于物理、化学、数学、生物、农业、医学、制药、航天、航空、气象、建筑、电子、通信、经济、商业、金融、证券、军事、交通、运输、管理等领域里的数值计算问题。

当前世界上应用的电子计算机基本上都是数字电子计算机，或称电子数字计算机，它与早期的模拟电子计算机相比，有其自身的一些特点。例如，它的工作方式是数字式的，这些数字的表示与运算都是离散的、二进制的（为了方便，若不作特别说明，本书后面所提到的“计算机”都指的是电子数字计算机）。还有，让计算机计算的问题都要求在空间和时间是有限的（比如要求解一个函数的极限值，严格说起来计算机是不可能求“极限”的，因为它不可能作无限次的运算，我们只能在某种意义下认为计算机可以求解）。如此等等这种的问题都是我们使用计算机时不可忽视的地方。也就是说，当我们利用计算机进行计算时，不得不适应它的这些特点。

为利用计算机来解决各种科学计算上的一些常见问题，需要注意以下几个问题。

1. 要考虑所要解决问题在数学理论上的条件与要求，选用有效合理的算法。

2. 求解问题的算法要考虑到计算机的特性。如计算机处理的数值对象必须是离散的，数值的取值范围必须是有限的等。

3. 采用先进的计算机程序分析、设计思想与方法。

4. 选用合适的计算机程序设计语言，将求解问题的算法在计算机上得以实现，求解出问题的解。

为此，本书收集了大量各类数值计算的有效算法，并附有详细的说明与解释；其次，本书采用了当前非常流行的面向对象的程序分析与设计的思想与方法，选用了能体现该思想方法，当前最为流行、最受企业界和学术界欢迎的程序设计语言之一——C++语言。书中算法及所实现的计算机程序都充分利用了C++语言所具有的各种特性，并有很多详细注释，使得读者在学习这些数值算法和程序，以及进行相关的编程、调试与维护上很方便，其学习与使用的效率都会有极大的提高。这样，读者从本书中不但可以熟悉科学计算的各种理论与算法，同时也可以学习面向对象的思想与方法，以及如何使用C++程序设计语言进行C++程序的编写，一举多得。

本书介绍的所有算法都在美国 Microsoft（微软）公司发布的 Visual C++ 6.0 环境中得到了实现，所有程序都可以在该环境中正常运行。本书光盘附有所有算法的相应Visual C++ 6.0 程序的示例，读者可以直接在Visual C++ 6.0 环境中对算法程序或示例程序进行修改、编译、连接、运行和调试。本章有专门的章节详尽地介绍了如何在Visual C++ 6.0 环境中建立所要求的工程，如何在该环境下对所附的C++程序进行修改、编译、连接、运行或调试，对于不熟悉在

Visual C++ 6.0 环境中编程的读者一定会有极大的帮助。

为方便读者，本书中各数值算法的程序都尽量采用了标准C++作为编程标准，以适应不同读者所应用的计算机软、硬件环境。书中程序是在 Windows 的 Visual C++6.0 环境下编写并通过的，如果读者需要，可以很容易地移植到其他计算机的软、硬件平台下。由于标准C++已经定义和提供了若干的标准规范的类库、模板等内容，因此本书将尽可能地采用这些已有的标准与规范。对于原标准中没有的部分，本书也另外作了一些补充，以方便程序的编写与应用。例如，有关矩阵的运算，标准C++没有定义相关的类，本书专门另外定义了矩阵类 matrix，以方便对于矩阵及与矩阵相关的算法（如线性方程组求解）的运算。因为整个程序框架仍然在标准C++框架中，所以它们只起着辅助的作用。

本章对某些需要特别注意的问题进行了专门的说明，以期使读者更好地了解与使用本书。

1.1 类　　库

C++是面向对象的计算机程序设计语言。面向对象程序设计方法所具有的封装、继承、多态等特性在C++中得到了充分的体现。这些特性的表现集中在其所定义的各种具体类上，而不同性质、不同功能的类的集合就形成了各种不同的类库。标准C++中已经定义了若干的类库。应用者灵活地使用这些类库，就可以高效、快速地构造出千千万万C++程序。因此，使用好C++的类库，是使用好C++语言的一个基本要求。本书的程序多次使用了标准C++提供的类库。

对于一些特别的问题，编程人员可以定义一些自己的类库，以方便问题的解决。由于在许多科学计算中要涉及到对大量矩阵的运算，而标准C++中没有对矩阵定义专门的类，处理起来不甚方便。为此，本书专门定义了以矩阵为对象的类 matrix，有关详细内容请参见本书的第 5 章“矩阵运算”，其中对类 matrix 做了非常详尽的介绍。

1.2 头文件及方法(函数)的命名

本书所有自定义的头文件和程序中应用的各个方法（函数）的命名都使用了与其相关的英文单词，或词组，或单词（词组）缩写的命名法，以方便读者对这些头文件和方法（函数）的理解与识别。

例如：LinearEquation. h 是求解线性方程组程序头文件的文件名；matrix. h 是矩阵模板类头文件的文件名。

又如：MatrixSymmetryRegular()是判断矩阵是否为对称正定矩阵的方法（函数）名；PolyValueOneDim()是求解一维实（复）多项式值的方法（函数）的方法（函数）名。

1.3 头文件的使用

本书对数值计算的各种算法进行了分类，每一类分别使用一章进行介绍。通常一个算法对应着一个C++程序中的方法（函数）。由于每一章都包含有若干个算法与程序，因此将每一章中所包含的这些算法程序都统一放在头文件中，以方便使用者调用。

每一章的算法程序都独立形成头文件。这些头文件通常分为两类：

1. 带有 .h 为文件扩展名的标准头文件。例如，LinearEquation.h 文件是关于求解线性方程组问题的算法程序的头文件，其中包含了若干求解线性方程组问题算法程序的方法(函数)原型。

2. 另一类是带有 .inl 为文件扩展名的头文件。它与前一个 .h 头文件配对出现，是其 .h 头文件中数值计算方法(函数)的定义(实现)。如与 LinearEquation.h 头文件配对的 .inl 头文件是 LinearEquation.inl，该文件中定义了各求解线性方程组问题算法程序方法(函数)的定义(实现)。在本节后面分别给出了 LinearEquation.h 头文件及部分 LinearEquation.inl 头文件的内容。

1.3.1 头文件 LinearEquation.h

```
//LinearEquation.h        线性方程组求解函数(方法)声明
// Ver 1.0.0.0.
// 版权所有(C) 何渝，2002
// 最后修改：2002.5.31

#ifndef _LINEAREQUATION_H
#define _LINEAREQUATION_H

#include <comm.h>              //公共头文件
#include <valarray>            //模板类 valarray 的标准头文件
#include <Matrix.h>            //矩阵类头文件

//全选主元高斯消元法
template <class _Ty>           //奇异返回 0，否则正常
int LE_TotalChoiceGauss(matrix<_Ty>& a, valarray<_Ty>& b);

//全选主元高斯—约当消元法
template <class _Ty>           //奇异返回 0，否则正常
int LE_TotalChoiceGaussJordan(matrix<_Ty>& a, matrix<_Ty>& b);

//三对角方程组的追赶法
template <class _Ty>
int LE_TridiagonalEquationGauss(valarray<_Ty>& b, valarray<_Ty>& d);

//  一般带型方程组求解
template <class _Ty>
int LE_StrapEquationGauss(matrix<_Ty>& b, matrix<_Ty>& d, int l, int il);

//对称方程组的分解法
template <class _Ty>
int LE_SymmetryEquation(matrix<_Ty>& a, matrix<_Ty>& c);

//对称正定方程组的平方根法
```

```
    template <class _Ty>
    int LE_SymmetryRegularEuationSquareRoot(matrix<_Ty>& a, matrix<_Ty>& d);

    //大型稀疏方程组全选主元高斯—约当(Gauss-Jordan)法
    template <class _Ty>
    int LE_SparseEuationTotalChoiceGaussJordan(matrix<_Ty>& a, valarray<_Ty>& b);

    //托伯利兹(Toeplitz)方程组的列文逊(Levinson)法
    template <class _Ty>
    int LE_ToeplitzEuationLevinson(valarray<_Ty>& t, valarray<_Ty>& b, valarray<_Ty>& x);

    //高斯-赛德尔迭代法
    template <class _Ty>
    int LE_GaussSeidelIteration(matrix<_Ty>& a, valarray<_Ty>& b, valarray<_Ty>& x, _Ty eps);

    //对称正定方程组的共轭梯度法
    template <class _Ty>
    int LE_SymmetryRegularEuationConjugateGradient(matrix<_Ty>& a, valarray<_Ty>& b, _Ty eps,
valarray<_Ty>& x);

    //线性最小二乘问题的豪斯荷尔德变换法
    template <class _Ty>
    int LE_LinearLeastSquareHouseholder(matrix<_Ty>& a, valarray<_Ty>& b, matrix<_Ty>& q);

    //线性最小二乘问题的广义逆法
    template <class _Ty>
    int LE_LinearLeastSquareGeneralizedInverse(matrix<_Ty>& a, valarray<_Ty>& b, matrix<_Ty>&
x, matrix<_Ty>& aa, _Ty eps,
    matrix<_Ty>& u, matrix<_Ty>& v);

    //病态方程组求解
    template <class _Ty>
    int LE_IllConditionedEquation(matrix<_Ty>& a, valarray<_Ty>& b, _Ty eps, matrix<_Ty>& x);

    #include <LinearEquation.inl>
    #endif //_LINEAREQUATION_H
```

1.3.2 头文件 LinearEquation.inl 节选

```
    //LinearEquation.inl                线性方程组求解函数(方法)定义
    // Ver 1.0.0.0
    // 版权所有(C) 何渝，2002
    // 最后修改：2002.7.31
```

```
#ifndef _LINEAREQUATION_INL
#define _LINEAREQUATION_INL

template <class _Ty>                    //全选主元高斯消去法
int LE_TotalChoiceGauss(matrix<_Ty>& a, valarray<_Ty>& b)
{
  long double MaxValue, tmp;            //记录主元绝对值
  int l(1), i, j, is;
  bool yn;

  int n = a.GetColNum();                //方程组阶数
  valarray<int> js(n);                  //保存换列位置

  for (int k = 0; k < n - 1; k++)       //全选主元
  {
     MaxValue = 0.0;                    //给保存主元绝对值变量赋初值
     for(i = k; i < n; i++)
        for(j = k; j < n; j++)
        {
           tmp = Abs(a(i, j));          //求 m(i,j)绝对值
           if(tmp > MaxValue)           //发现一个更大的主元
           {
              MaxValue = tmp;           //保存新主元绝对值
              js[k] = j;                //新主元所在列
              is = i;                   //新主元所在行
           }
        }
  yn = FloatEqual(MaxValue, 0);
  if(yn) l = 0;                         //主元为 0
  else
  {
     if(js[k] != k)                     //换列
        for(i = 0; i < n; i++) swap(a(i, k), a(i, js[k]));
     if(is != k)                        //换行
     {
        for (j = k; j < n; j++)swap(a(k, j), a(is, j));
        swap(b[k], b[is]);              //方程组右边第 k 元素与第 is 元素交换
     }
  }
  if(l == 0)                            //矩阵奇异(主元为 0)
  {
     printf("fail 1\n");
     return 0;                          // 求解失败,返回 0 值
  }
```

```
    MaxValue = Abs(a(k, k));
    for(j = k + 1; j < n; j + + )a(k, j) / = a(k, k);     //MaxValue;
        b[k] / = a(k, k);                                  //MaxValue;
        for(i = k + 1; i < n; i + + )
        {
            for(j = k + 1; j < n; j + + )
            {
               a(i, j) = a(i, j) - a(i, k) * a(k, j);
            }
            b[i] = b[i] - a(i, k) * b[k];
        }
    }
    MaxValue = Abs(a((n - 1), (n - 1)));                   //主元
    yn = FloatEqual(MaxValue, 0);
    if(yn)                                                 //主元为 0
    {
         cout<<"fail 2"<<endl;
         return(0);                                        //求解失败,返回 0 值
     }
     b[n - 1] / = a((n - 1), (n - 1));                     //求解方程组右边 X 的解
     for(i = n - 2; i >= 0; i - - )                        //回代过程
     {
        _Ty t = 0.0;
        for(j = i + 1; j < n; j + + )t = t + a(i, j) * b[j];
        b[i] = b[i] - t;
     }
     js[n - 1] = n - 1;                                    //X 最后一个元素不用换
     for(k = n - 2; k >= 0; k - - )                        //k 可以从 n-2 开始
         if(js[k] ! = k)                                   //交换 X 的元素位置(由全选换列产生的)
             swap(b[k], b[js[k]]);
     return(1);                                            //方程组求解成功!
}
.....
.....
#endif //_LINEAREQUATION_INL
```

1.4 模　　板

利用C++提供的模板特性,会给程序的编写,以及今后程序的使用都带来极大的方便。本书主要使用了函数模板。对于模板的使用,本书对数据类型处理的程序就是一个很好的例子。

通常计算机程序设计语言对要处理的数据,在进行输入输出时都要求对这些数据指定不同的数据类型,这样的要求是计算机本身特性所决定的。比如,数据是按整型数据进行处理,

还是按浮点型数据进行处理？即，对于同样的整数数据“10”，用整型数 10 表示，还是用浮点型数 10.0 表示，这在计算机内部的处理(运算和存储)是完全不一样的。

另外，对要进行处理的数据仅仅标出它是整型或浮点型还不够，还要求有更细的划分与指定。例如，如果对前面的数据“10”指定为浮点型“10.0”的数据类型，还需要进一步指明是单精度(float)类型的，还是双精度(double)类型的？甚至可能是长双精度类型(long double)的，如此等等。因此，在编程时要特别注意这些细节，否则就可能给计算带来问题，产生隐患，有时甚至会给程序的运行造成致命的后果。

鉴于上述原因，为了避免这些看似繁琐的编程细节给编程人员带来不便，C++提供了“模板”(template)功能，利用它可以有效地避免或缓解这类问题。下面的小例子用来说明函数模板的应用，其他情况类似，读者可以通过该小例子举一反三地应用该特性。

下面给出的是本书中存在的三行C++源程序代码：

```
//取 x 符号，+ - 或 0
template <class T>
T Sgn(const T& x);
```

这三行代码是为求数值数据 x 的符号面编写的符号函数的原型代码，第一行是注解行，真正的代码是后两行。在第三行中声明了函数的函数名为 Sgn，形式参数为 x。参数 x 的数据类型为 T，这个 T 在第二行代码中进行了声明。

由此，当需要调用 Sgn()函数去获得某一个实际参数 a 的符号时，实际参数 a 的数据类型就可以是各种各样的类型：可以是整型(int)的，也可以是浮点型(float)的；若是整型的，既可以是短整型(short int)的，也可以是长整型(long int)的；若是浮点型的，既可以是单精度型(float)的，也可以是双精度型(double)的，还可以是长双精度型(long double)的。这就是说，当编程人员要调用 Sgn()函数时，无论实际参数 a 是这些不同的数据类型的哪一种数据，都同样可以得到其符号。

再来看另一个小例子：

```
//取 x 绝对值
template <class T>
long double Abs(const T& x);
```

这是作者自定义的、用来求数值数据 x 的绝对值所编写的函数原型代码。在代码的第三行中声明了函数的函数名为 Abs，形式参数为 x。参数 x 的数据类型为 T，这个 T 在第二行代码中进行了声明。函数 Abs()的参数 a 的数据类型既可以是整型的，也可以是浮点型的，甚至可以是复数类型的。

这要说明一点的是，在调用函数 Abs()时，如果它的实际参数 a 的数据类型是复数，对复数 a 所求的绝对值来说，实际上函数 Abs()是求复数 a 的“模”。也就是说，该 Abs()函数将求实数绝对值的概念扩大到了复数。这时，这个自定义的 Abs()函数要使用到C++中专门定义的一个类——complex(复数类)，有关 complex 类的情况，请参见本书的第 2 章“复数运算”，其中有详细的介绍。而且，complex 复数类型数据既可以是整型的，也可以是浮点型的，函数 Abs()对它们都有效。

函数 Sgn()和 Abs()的定义在公共头文件 comm.h 中，有关它们的内容及实际应用的示例，在本章 1.7 节“公共头文件”中给出。

1.5 重　　载

重载也是C++中一个很重要的特性，灵活地应用该特性将会给编程带来许多意想不到的效果。重载主要有对函数的重载和对运算符的重载。本书中的程序有不少的这类应用。

1.5.1 重载函数

C++允许在其程序中拥有多个重名函数，这种函数称为重载函数。重载函数的函数名都相同，但可能其参数的个数不同；也可能参数个数相同，但参数的类型不同。这样，在调用这种函数时，可以起到不同的作用，实现不同的功能，获得不同的结果。

例如，在头文件 comm.h 中有这样的代码行：

```
//打印数组(向量)所有元素值
 template <class T>
 void ValarrayPrint(const valarray<T>& vOut);

//打印某个指定数组(向量)元素值
template <class T>
void ValarrayPrint(const valarray<T>& vOut, size_t sPosition);
```

两个 ValarrayPrint()函数，它们的名字都相同，但参数不一样，第一个仅有一个参数，而第二个有两个参数。这两个函数都执行打印向量值的功能，第一个函数的功能是一次将向量的所有元素都打印出来，而第二个函数的功能是一次仅打印某一个指定的向量元素。根据不同的情况，读者可以灵活地选用不同的函数，完成不同的功能。

关于 ValarrayPrint()这两个函数的应用示例请参见本章 1.7 节“公共头文件”中的 1.7.2 “示例”，其他的重载函数请参见有关章节。

1.5.2 重载运算符

C++允许对已经定义过的运算符进行再定义，以满足某些特别的需要。举例来说，对于变量 A 和 B，可以进行各种算术运算，如＋、－、＊，如果这两变量是普通的整型或浮点型变量，这些运算已经定义好了，可以直接应用。但如果它们是矩阵变量，要想进行矩阵之间，或者矩阵与普通数值(如整型或浮点型)之间的＋、－、＊运算，那原来定义好的这些运算符将不适用于它们。

由于在本书中专门定义了矩阵类 matrix，并在该类中又对诸如＋、－、＊这样的许多运算符进行了专门的定义(重载)，因此，当变量 A 和 B 是矩阵类 matrix 的对象实例时，对它们就可以直接应用＋、－、＊这样的运算符，进行＋、－、＊的运算。这样的运算可以是矩阵与矩阵之间的，也可以是矩阵对数值(如整型或浮点型)之间的。这样将给编程人员带来了极大的方便，减少许多不必要的麻烦，大大提高工作效率。

关于矩阵类 matrix 具体应用的实例，请参见本书第 5 章“矩阵运算”，其中有非常详细的介绍及众多的示例。

1.6 浮点数的比较

在数值计算中应用最多的数据是浮点数。前面已经提到，在计算机中对浮点数的处理与对整型数的处理是不一样的。也就是说，浮点型数据与整型数据在计算机中的运算与存储是不同

的(比如,对于CPU来说,有专门处理浮点数据的CPU或专门的处理单元)。因此,在对待这两类数据时,就要有所区别。典型的问题是浮点数的比较,由于在许多科技书籍中,以及在许多现在流行的程序和软件中,对此处理都出现了不同程度的问题,因此在这进行特别的说明。

假设,变量 x 为浮点类型,且它已经被赋予某数值,当要用它与数据0进行比较时,若有下面的C++程序代码:

```
......
float x;
......
if(x = =0).....          //注意
```

注意,在//注意这一行中的语句 if(x==0)…… 中,如果变量 x 是整数型的变量,那么与0的比较没有问题,但在这里,x 是浮点类型的,则变量 x 一般是不应该直接与0进行比较的。同样类似的问题还有 if(x! =0.0)……,if((x-a)==b)……等。

这类问题,不仅在C++语言中存在,在所有要涉及到浮点数运算的计算机程序设计语言中都存在。碰到这样的问题时,应该特别注意。

对这类问题,本书中是这样处理的:首先求浮点类型变量 x 的绝对值,若这个绝对值小于某一个很小很小的正数,比如 10^{-6} 或 10^{-30} 等,就认为变量 x 为0了。这个很小很小的正数通常都可以事先设置好,其数值的大小与具体计算的要求有关,说到底,还与计算机所具有的位长有关。

为了合理有效地解决这个问题,作者编制了相关的比较函数 FloatEqual()和 FloatNotEqual()专门处理这类问题,它们可以适应不同的浮点数类型数据,本章1.7节"公共头文件"中有其示例。

1.7 公共头文件

在本书中涉及到成百个常用数值计算问题的算法与程序,在所有这些程序中存在着一些具有共性的问题。为了方便处理这些具有共性的问题,专门编制了公共头文件。

公共头文件有两个:comm.h 和 comm.inl。前者包括公共宏的定义及公共函数的原型,后者包括公共函数的定义(实现)。

在公共头文件中所定义的宏和函数,其应用不限于本书所包含的程序,读者也可以应用到自己的程序中。

公共头文件中包括的宏定义常数有:

```
const float FLOATERROR = 1.0e-6F;
const double DOUBLEERROR = 1.0e-15;
const long double LONGDOUBLEERROR = 1.0e-30;
const double GoldNo = 0.618033399;     //黄金分割常数
```

前三个常数可以在程序迭代判断误差时使用,也可用作区间分割的步长等地方;第四个常数作为程序中使用到黄金分割算法时的黄金分割常数。

公共头文件中包括的公共函数有:

- 条件编译的异常处理函数 Assert();
- 求绝对值的函数 Abs();
- 求符号的函数 Sgn();
- 比较浮点数相等与不相等的函数 FloatEqual()和 FloatNotEqual();

- 求最小值和最大值的函数 Min()和 Max()；
- 打印数组(向量)所有元素值和打印指定数组(向量)元素值的函数 ValarrayPrint()。

1.7.1 comm.h

```
// Comm.h                                公共头文件
// Ver 1.0.0.0
// 版权所有(C) 何渝，2002
// 最后修改：2002.5.31

#ifndef _COMM_H    //避免多次编译
#define _COMM_H

/******
    由于 NDEBUG 是为程序调试(debug)期间使用的，当调试期完毕，程序出发行版(release)后，NDEBUG 将失去作用。为了能够使 assert()函数在发行版中也可以使用，则定义了下面的条件编译 NDEBUG，意思如下。
    如果已经定义 NDEBUG，则定义宏 Assert(x)：当 x 为假时，执行 throw，交由系统已定义的方式处理。否则将 Assert(x)定义为函数 assert(x)，该函数对参数 x 进行测试，x 为假时，函数终止程序并打印错误信息。
*****/
#ifdef NDEBUG
  #define Assert(x)          if (! x)throw;
#else                                  //否则用系统定义的函数 assert()处理
   #include <cassert>
   #define Assert(x)         assert(x);
#endif                                 //NDEBUG

#include <complex>                     //模板类 complex 的标准头文件
#include <valarray>                    //模板类 valarray 的标准头文件
using namespace std;                   //名字空间

const float FLOATERROR = 1.0e-6F;
 const double DOUBLEERROR = 1.0e-15;
 const long double LONGDOUBLEERROR = 1.0e-30;

const double GoldNo = 0.618033399;  //黄金分割常数

//取 x 绝对值
template <class T>
long double Abs(const T& x);

//取 x 符号，+、-或 0
template <class T>
T Sgn(const T& x);
```

```
//比较两 float 浮点数相等
bool FloatEqual(float lhs, float rhs);
//比较两 float 浮点数不相等
bool FloatNotEqual(float lhs, float rhs);
//比较两 double 浮点数相等
bool FloatEqual(double lhs, double rhs);
//比较两 double 浮点数不相等
bool FloatNotEqual(double lhs, double rhs);
//比较两 long double 浮点数相等
bool FloatEqual(long double lhs, long double rhs);
//比较两 long double 浮点数不相等
bool FloatNotEqual(long double lhs, long double rhs);

//求 x 与 y 的最小值,返回小者
templatc <class T>
T Min(const T& x, const T& y);

//求 x 与 y 的最大值,返回大者
template <class T>
T Max(const T& x, const T& y);

//打印数组(向量)所有元素值
template <class T>
void ValarrayPrint(const valarray<T>& vOut);

//打印某个指定数组(向量)元素值
template <class T>
void ValarrayPrint(const valarray<T>& vOut, size_t sPosition);

#include "Comm.inl"//实现

#endif// _COMM_H
```

1.7.2 示例

为了便于理解和使用公共头文件中定义的宏和函数,下面给出一个C++的示例程序,其主函数及程序如下:

```
//Comm.cpp                    使用公共头文件示例

#include <iostream>           //输入输出流头文件
#include "Comm.h"             //公共部分头文件
#include "Matrix.h"           //矩阵类及相关函数等的定义
using namespace std;          //名字空间
```

```
// 定义控制台应用程序的入口点
void main()
{
  int ia(10), ib(-23);
  double fx(1.5369200000000001), fy(-38.67);
  complex<double> cf(fx, fy);

  //显示初始数据
  cout << "初始数据:" << endl;
  cout << " ia = " << ia << endl;
  cout << " ib = " << ib << endl;
  printf(" fx = %25.20f\n", fx);
  printf(" fy = %25.20f \n", fy);
  cout << " cf = " << cf << endl << endl;

  //取绝对值运算
  cout << "取绝对值运算:" << endl;
  cout << " Abs(ia) = " << Abs(ia) << endl;
  cout << " Abs(ib) = " << Abs(ib) << endl;
  cout << " Abs(fx) = " << Abs(fx) << endl;
  cout << " Abs(fy) = " << Abs(fy) << endl;
  cout << " Abs(cf) = " << Abs(cf) << endl << endl;

  //取符号运算
  cout << "取符号运算:" << endl;
  cout << " Sgn(ia) = " << Sgn(ia) << endl;
  cout << " Sgn(ib) = " << Sgn(ib) << endl;
  cout << " Sgn(fx) = " << Sgn(fx) << endl;
  cout << " Sgn(fy) = " << Sgn(fy) << endl << endl;

  //浮点数比较
  cout << "浮点数比较:" << endl;
  if(FloatEqual(fx,1.53692))
      printf(" fx = = 1.53692\t fx = %25.20f \n\n", fx); else
      printf(" fx ! = 1.53692 \t fx = %25.20f\n\n", fx);

  if(FloatNotEqual(fy,-38.671))
      printf(" fy ! = -38.671 \t fy = %25.20f \n\n", fy);
  else
      printf(" fy = = -38.671 \t fy = %25.20f \n\n", fy);

  //求最小值运算
  cout << "求最小值运算:" << endl;
  cout << " Min(ia, ib) = " << Min(ia, ib) << endl << endl;
```

```
    //求最大值运算
    cout << "求最大值运算:" << endl;
    cout << " Max(fx, fy) = " << Max(fx, fy) << endl << endl;

    double a[4] = {1.01, 1.02, 1.03, 2.04};

    valarray<double> va(a, 4);

    //打印数组(向量)值
    cout << "打印数组(向量)值:" << endl;
    ValarrayPrint(va);
    cout << endl;

    //打印某个指定数组(向量)元素值
    cout << "打印某个指定数组(向量)元素值:" << endl;
    for(ia = 0; ia<4; ia + + )
        ValarrayPrint(va, ia);
}
```

示例程序的运行结果如下。

```
初始数据:
  ia = 10
  ib = -23
  fx = 1.53692000000000010000
  fy = -38.67000000000000200000
  cf = (1.53692, -38.67)

取绝对值运算:
  Abs(ia) = 10
  Abs(ib) = 23
  Abs(fx) = 1.53692
  Abs(fy) = 38.67
  Abs(cf) = 38.7005

取符号运算:
  Sgn(ia) = 1
  Sgn(ib) = -1
  Sgn(fx) = 1
  Sgn(fy) = -1

浮点数比较:
  fx = = 1.53692      fx = 1.53692000000000010000

  fy ! = -38.671      fy = -38.67000000000000200000
```

```
求最小值运算：
  Min(ia, ib) = -23

求最大值运算：
Max(fx, fy) = 1.53692

打印数组(向量)值：
        1.01        1.02        1.03        2.04

打印某个指定数组(向量)元素值：
        1.01
        1.02
        1.03
        2.04
```

1.8 Visual C++ 6.0 的编译运行环境

本书所有数值算法的C++程序都是在美国 Microsoft(微软)公司发布的 Visual C++ 6.0 (简称 VC6.0)环境下进行编写、编译和连接的，最后生成可执行程序。所有这些程序的C++源程序 .cpp 文件、可执行 .exe 文件和运行结果数据 .dat 文件都保存在本书所附的光盘中，读者可以调用。

VC6.0 是一个运行在 Windows 环境下，非常庞大的C++编译系统，功能十分强大。由于本书所涉及的程序对输入输出的界面要求不高，一方面可以不把过多的时间花费在界面的编制、设计与调试上，另一方面，如果需要将程序移植到其他非 PC 机结构、非 Windows 环境下，也更方便，所以本书所有程序都采用控制台方式，这样编制的C++程序将相对简单得多，也可以节省很多消耗在程序界面上的精力与时间。该方式的操作大部分是在 VC 6.0 的 Windows 环境的图形界面下，程序的编辑、编译、修改、测试等操作都在 Windows 环境中，仅仅是程序的运行这一步骤是在模拟状态的 DOS 下，而其他所有编程的环节都仍是在 Windows 环境里，程序的调试运行结束后又可以返回 Windows 环境，这对程序的整个编制调试过程非常方便，不会有多大的影响。生成的.exe 可执行文件(程序)，须在 DOS 环境下运行。

为了方便读者进行学习和应用，本节特别简要地介绍一下该系统的编译运行环境，以及如何进行编程调试工作，读者可根据自己的具体情况灵活处理。若读者对 VC6.0 环境已经很熟悉，可以跳过此节。

1.8.1 VC6.0 工程(文件夹)的创建

要在 VC6.0 下进行编程，需要事先做一些准备工作，然后才能进到编程阶段。

首先，对要编制的程序需要建立一个工程(Project)。在建立这个工程的过程中，VC6.0 需要一个专门与该工程相配套的文件夹，它被用来存放编程调试过程中所需要的各种文件，其中包括有 VC6.0 系统自身需要的若干文件和专用文件夹，也包括用户在编程过程中所编制程序要生成的各C++源文件、编译文件、可执行文件等，以便做相应的处理工作。这些是 VC6.0

所必需的。使用者初次使用时似乎会感到繁琐，但创建工程的整个过程，在创建工程向导的程序引导下，只需几个步骤就可以完成。

在建立一个工程过程中所需要的文件夹可以是事先建立好(已存在的)，也可以在第一次调用 VC6.0 后在建立新工程的过程中生成的。这个工程，以及相应的文件夹，一旦建立好之后，就可以长期使用，不必每次使用该工程时再重新建立。

用户可以根据自己的实际具体情况，一个程序建立一个这样的工程(文件夹)，也可以多个程序建立一个工程，共用一个它拥有的文件夹。

由于本书中包括的程序很多，建议统一先建立一个总的文件夹(如这个总文件夹可命名为 NumComp)，然后在这个统一的总文件夹下再建立以章为单位的多个工程。然后，在建立工程的过程中将生成的对应文件夹(如对第 5 章的程序可建立的工程及文件夹命名为 Chap5)作为总文件夹下面的子文件夹，以方便统一管理。每一章所包含的程序可以都放在各自的子文件夹中(如第 5 章中包含的所有的C++源程序 .cpp 文件)，以便进行统一的程序编制与调试。

在总文件夹下还应再另建立一个包含所有自编制的头文件(包括本书提供的各个头文件)的文件夹(如命名为 include)，其中包括有各章程序都要使用的公共头文件，如 Comm.h，也包括各章独立使用的头文件，如第 16 章“特殊函数”中程序要使用的头文件 SpecialFunction.h，以方便统一管理与使用。

下面以建立第 5 章“矩阵运算”的工程与对应的子文件夹为例进行介绍，工程及文件夹的名字命名为“Chap5”。假设将整个工程及文件夹都放在计算机的 E 盘里，总文件夹 NumComp 已经存在，建立在 E 盘的根目录上。假设 Chap5 文件夹不存在，启动 VC6.0 后，在建立 Chap5 工程的过程中将会由 VC6.0 自动生成该文件夹。

创建 Chap5 工程的整个过程步骤如下，其他工程的建立也类似，读者可模仿该步骤自己创建所需工程。

1. 启动 VC6.0，弹出 VC6.0 的主窗口，参见图 1.1。

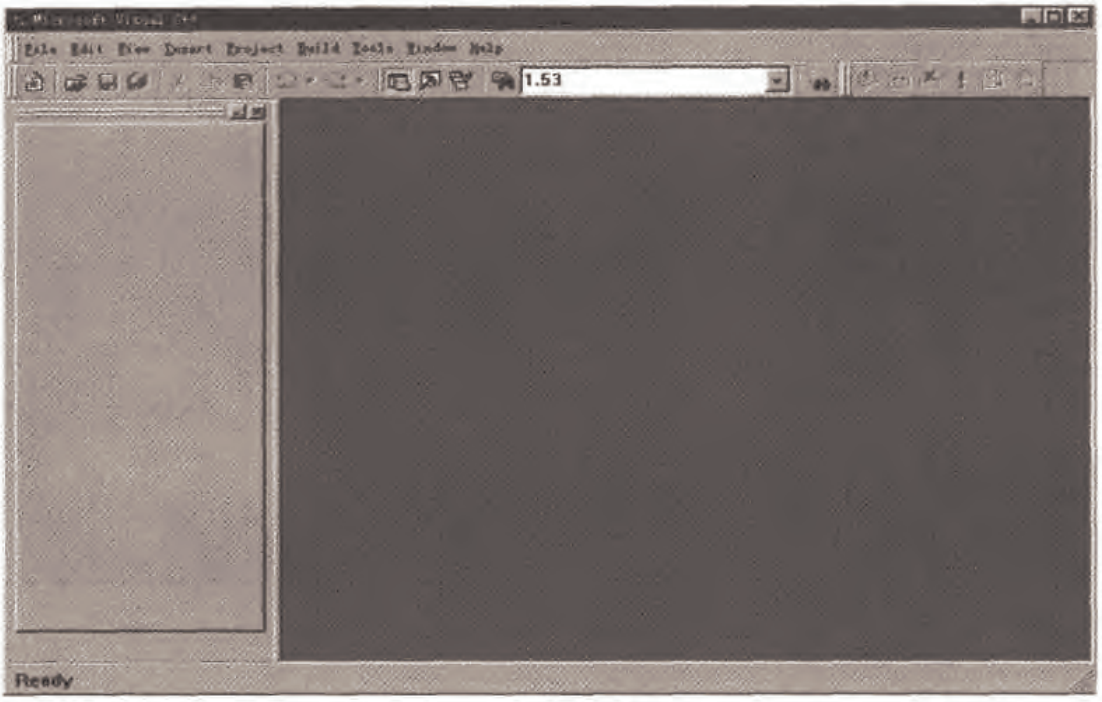


图 1.1　VC6.0 启动后的主窗口界面

2. 选择“File”菜单下的“New...”命令项，弹出“New”对话框，参见图 1.2。

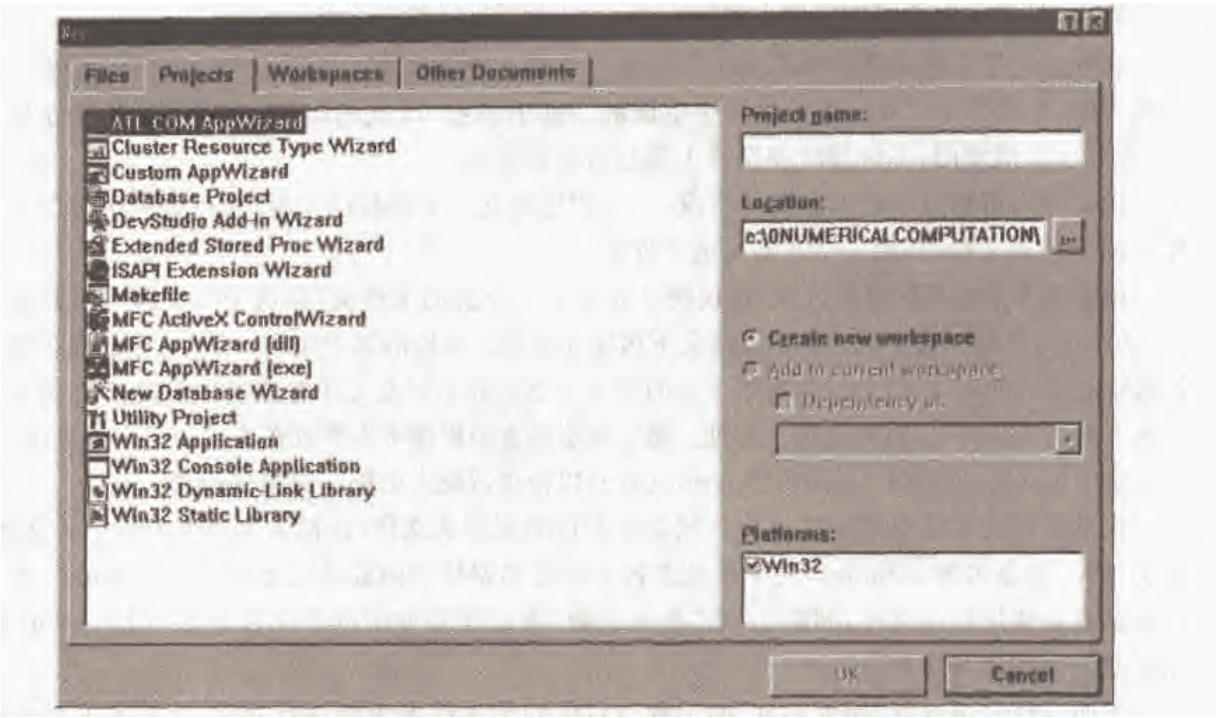


图 1.2　New 对话框

3. 选择“Project”标签中的“Win32 Console Application”选项，然后在右边“Location:”文本框中选中“e\NumComp\”文件夹，在“Project name:”文本框中键入“Chap5”，参见图 1.3。



图 1.3　在 New 对话框中建立 Chap5 工程

4. 点击对话框右下方“OK”按钮，弹出“Win32 Console Application － Step 1 of 1”对话框，参见图 1.4。

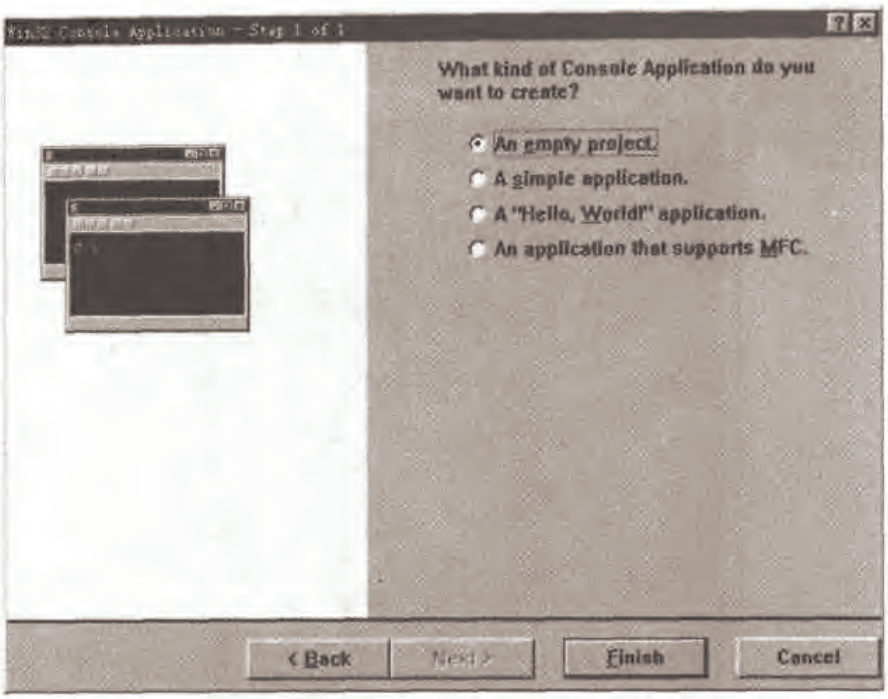


图 1.4　Win32 Console Application — Step 1 of 1 对话框

5. 选择“An empty project”单选钮(缺省),点击“Finish”按钮,弹出“New Project Information”对话框,参见图 1.5。

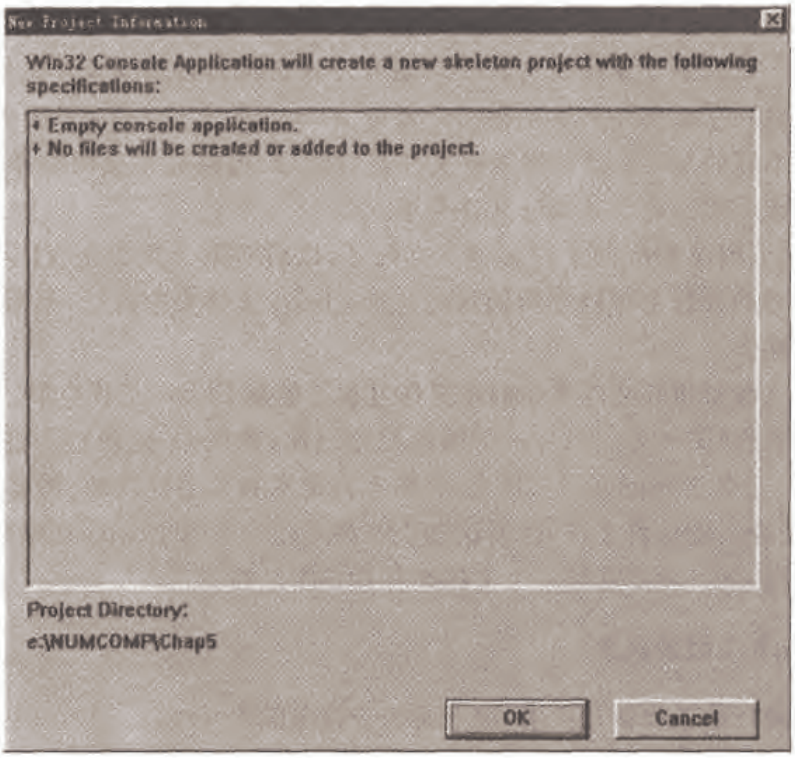


图 1.5　New Project Information 对话框

6. 点击“OK”按钮，返回 VC6.0 主窗口。这时主窗口的标题栏上将出现“Chap5 - Microsoft Visual C++”字样，参见图 1.6。

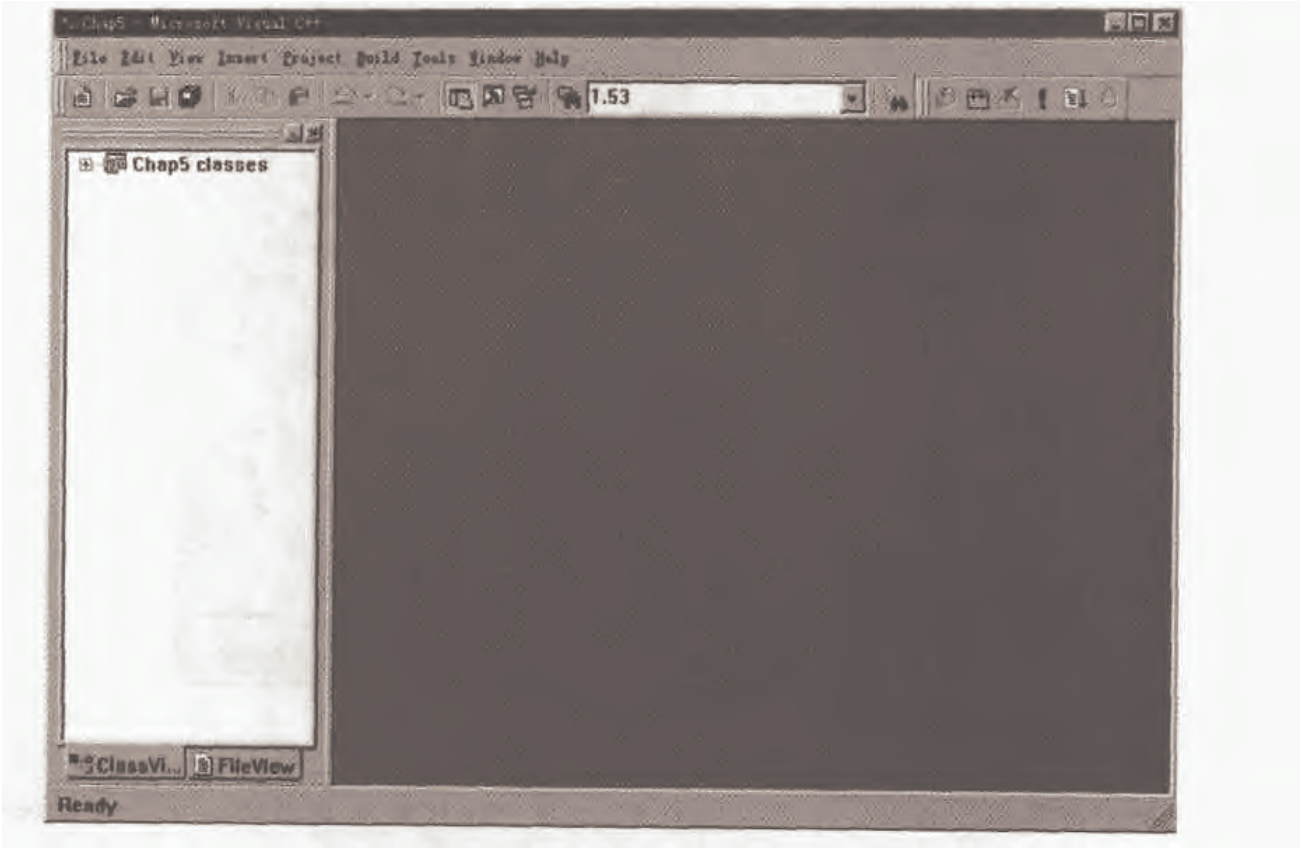


图 1.6 VC6.0 的 Chap5 主窗口

7. 查看 E 盘，将看到“E:\NumComp\Chap5”文件夹。在该文件夹下有三个新的文件，不用移动这些文件。以后第 5 章中所有的C++源程序文件都可以放在这个文件夹中，统一进行各种处理工作。另外在该文件夹里还有一个名为“debug”的文件夹，它里面也有若干文件，也不用移动它们。当以后对第 5 章的源程序文件进行编译、连接后生成的编译文件和可执行文件都将在自动生成后放在这个 debug 文件夹里。

至此，Chap5 工程及相应的文件夹就建立好了，以后对第 5 章中所有C++程序需要进行编制、修改、调试等的操作都可以在所创建的这个 Chap5 文件夹中进行，不需要再一次进行本节前面所叙述的操作。

其他章节的工程及相应文件夹的创建操作过程与创建 Chap5 工程类似。

在创建完成工程(文件夹)之后，进行编制程序之前，建议：一定还应当在总文件夹 NumComp 下要建立一个名为“include”文件夹，将所有自定义和本书提供的、需要用到的头文件都放到里面，以方便今后的编程工作，因为这些头文件不止一个像 Chap5 这样的工程使用，其他工程也可能要使用，统一建立这样一个文件夹有其方便之处。

1.8.2 VC6.0 工程的设置

在正式进行编程前还要做几件事，用下面的步骤来说明。

1. 点击 VC6.0 主窗口中的“Tools”菜单，选择“Options...”命令项，弹出“Options”对话框，参见图 1.7。

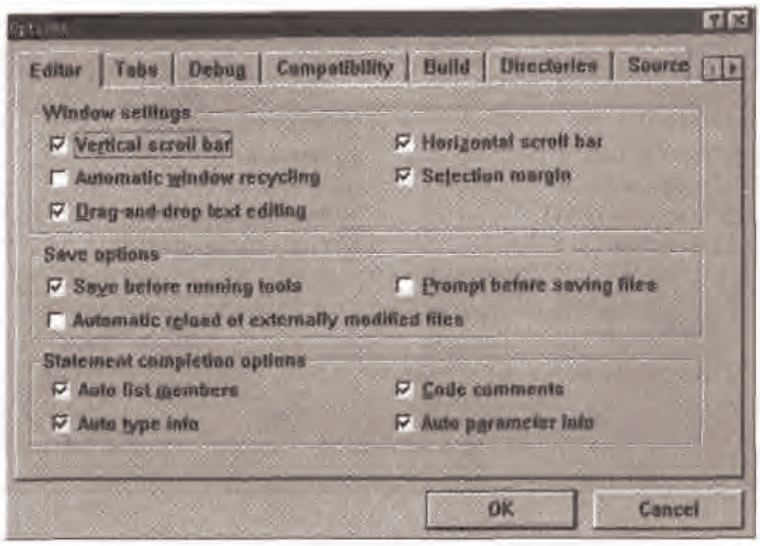


图 1.7　Options 对话框

2. 在“Options”对话框里选择“Directories”标签，参见图 1.8，双击该标签下方“Directories:”框里最下方的虚框，在变化后的文本长框里键入“E:\NumComp\include”，回车，再点击“Options”对话框最下方的“OK”按钮，返回到 VC6.0 主窗口。

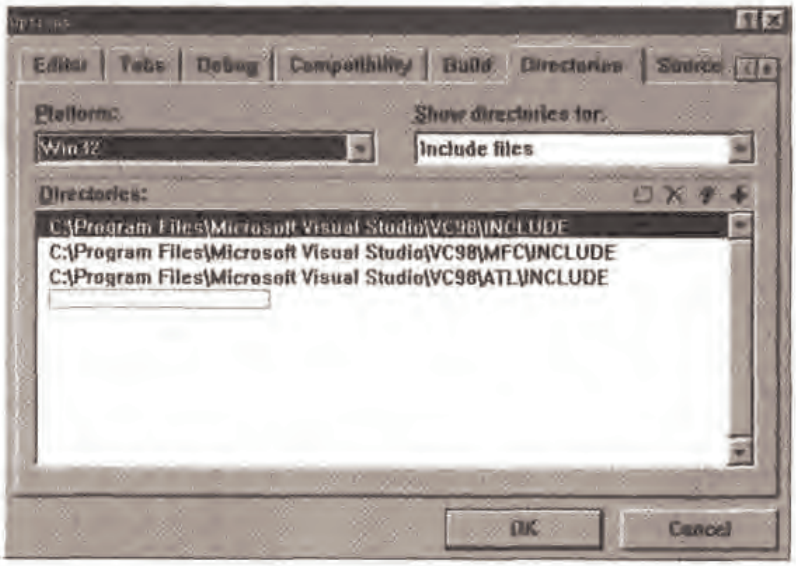


图 1.8　Options 对话框中的 Directories 标签

这是为了让 VC6.0 系统保存读者自编头文件和本书提供的头文件的文件夹路径，以便在编制与调试程序时，系统能够找到所需要的这些头文件。该操作只需要一次，以后不必再重复。

3. 在 VC6.0 主窗口中点击左下角的“FileView”标签，参见图 1.6，在该标签中展开“Chap5 files”。

4. 在“Header Files”条幅上按下鼠标右键，在弹出的快捷菜单中选择“Add Files to Folder...”命令项，在弹出的“Inset Files into Project”对话框中找到头文件所在文件夹位置（如 E:\NumComp\include），选择需要进行显示、编制或修改的头文件，再点击“OK”按钮，参见图 1.9。

5. 在“Source Files”条幅上按下鼠标右键，在弹出的快捷菜单中，选择“Add Files to Folder...”命令项，在弹出的“Inset Files into Project”对话框中找到源程序文件所在文件夹位

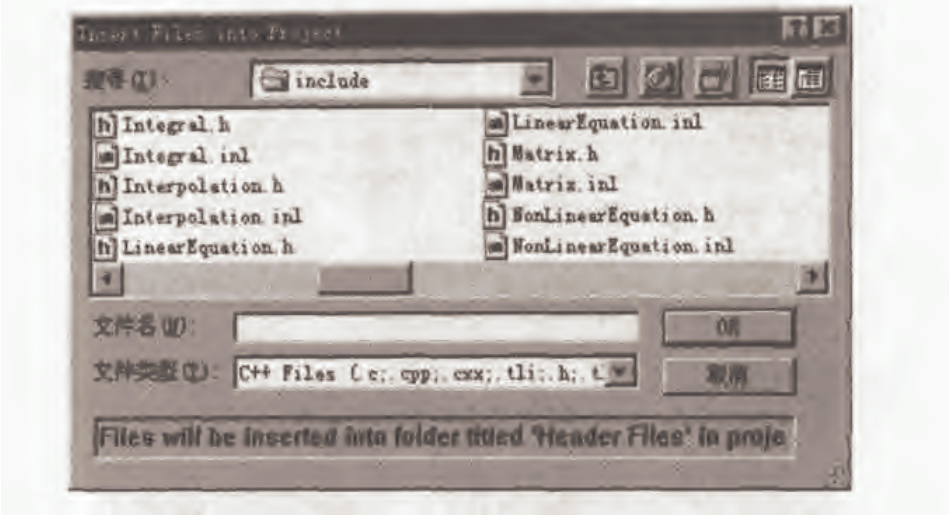


图 1.9　Inset Files into Project 对话框

置(如 E:\NumComp\Chap5),选择需要进行显示、编制或修改的头文件,再点击"OK"按钮。对话框内容类似图 1.9。

6. 展开"Source Files"条幅和"Header Files"条幅,点击其中的某一文件名,该文件的内容中会显示在 VC6.0 主窗口的右边编制窗框中。图 1.10 所示的是"MatrixLU.cpp"源程序文件的内容。

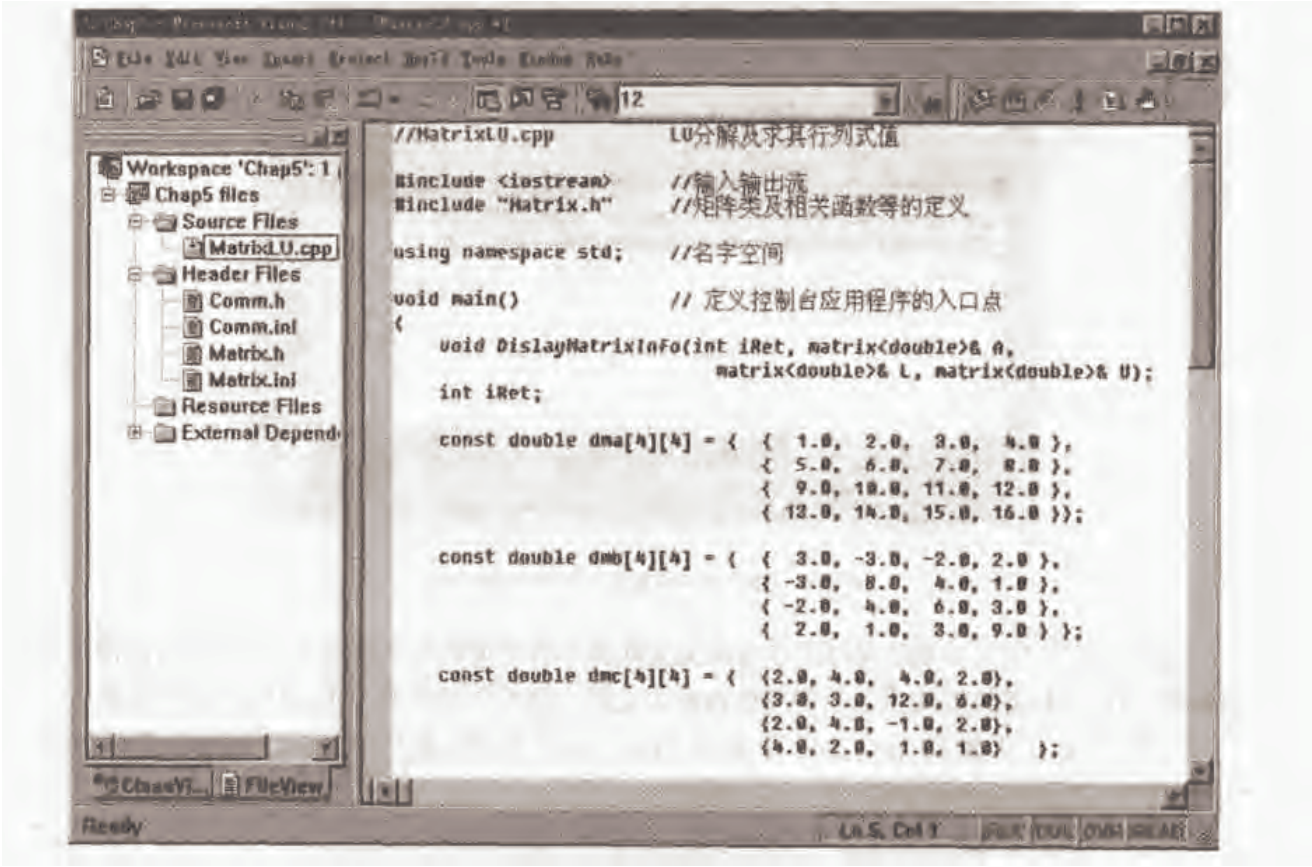


图 1.10　VC6.0 主窗口的程序编制调试界面

1.8.3　VC6.0 程序编制调试

在前面的工作完成后,就可以进行程序的编制、修改、调试了。若是第一次生成新的源程序,

可以在 VC6.0 的主窗口中新生成一页，编写完必要的源程序代码后，将其保存在相应文件夹中。

如果要对C++程序进行编译，可以使用所带菜单项里的命令项，也可以使用工具栏中的工具图标，还可以使用该命令的键盘快捷命令。其他的如连接、运行等操作都类似。执行这些操作后，会生成一些相应的信息，这些信息将出现在最下方的信息窗格中，为编程人员提供必要的参考，参见图 1.11。

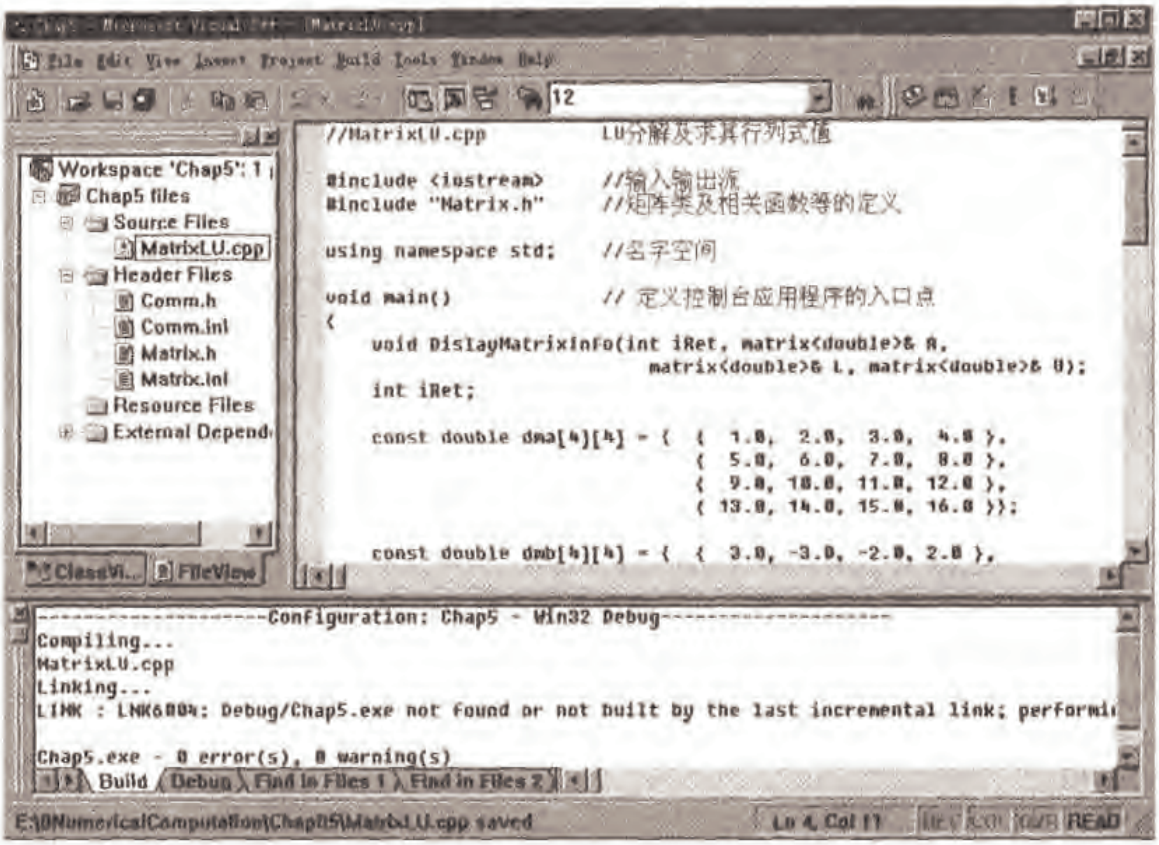


图 1.11 VC6.0 主窗口的程序编译后的界面

1.8.4 注意事项

1. 在 VC6.0 环境下编写、调试完一个程序后，若要对该工程（其文件夹里的）另一个程序进行编译、连接等操作，最好将“Source Files”条幅下的源程序文件去掉，改成另一个，特别是带有 main()主函数的源程序文件。这是因为，当“Source Files”条幅下有好几个源程序文件时，如果都带有 main()主函数，在编译时可以分别编译，但在连接时，VC6.0 系统将报错。这样做会给编程带来一点点不方便，但省去了在一个源程序文件中要创建一个工程的麻烦。

下面就以第 5 章所包含的程序作为例子进行说明。第 5 章的工程 Chap5 文件夹中包含有十几个算法程序，如 MatrixExample.cpp，MatrixDeterminant.cpp，MatrixCircumPrint.cpp 等。如果第一次对 MatrixExample.cpp 程序进行编制、编译等操作，在完成所有的操作后，要改成对第二个程序 MatrixDeterminant.cpp 进行编制、编译等操作，就必须将 MatrixExample.cpp 从“Source Files”条幅下的源程序文件中去掉，改换成第二个程序 MatrixDeterminant.cpp，否则 VC6.0 系统会报错。这样做似乎有些麻烦，但带来的好处是不需要对每一个程序分别都建立一个工程（以及生成各自的诸文件夹），这一章中的所有程序都集中在一个文件夹中，它们相互之间都有联系，可以统一进行编制、调试。

2. 由于一个工程中包含有多个程序，而每个程序都使用同一个工程的名字，因此生成的可执行文件都相同。下面以第 5 章的工程 Chap5 为例来说明，该工程名为 Chap5，在该文件夹下的任何一个程序，编译连接后生成的可执行文件，其文件名都为 Chap5.exe。也即，当对程序 MatrixExample.cpp 编译连接成功后，生成的可执行文件名为 Chap5.exe；换一个程序 MatrixDeterminant.cpp，在编译连接成功后，生成的可执行文件名仍为 Chap5.exe，因为它们用的都是同一个工程。如果需要区别，可以在某一程序生成 Chap5.exe 可执行文件后，将该文件名进行修改，换成另一个文件名，如 MatrixDeterminant.exe。下次对另一个程序进行编译连接时，又会生成另一个 Chap5.exe 可执行文件，如需要，再将这个 Chap5.exe 进行改名。

1.9 输　　出

本书所有示例中的输出都是将运行结果(数据与信息)或者运算过程中出现的问题提示信息输出至计算机的显示(监视)器上。如果需要将这些运算结果或运算过程中出现的问题提示输出至计算机磁盘中，以计算机文件的形式保存起来，可以采用下面介绍的两个方法中的任一种，第二方法也可以直接将程序运行结果输出至打印机。

1.9.1 在程序中加特别的输出语句

为将程序执行后生成的运算结果或者运算过程中出现的问题提示输出至计算机磁盘中，以计算机文件的形式保存起来，可以在程序中修改或添加相应的输出语句。参见下面的例子。

在C++程序中，一般都使用如下的命令和语句进行信息输出：

```
#include <iostream>              //输入输出流头文件
……
main()
{
   ……
   cout <<…… <<……;              //输出信息至显示器
   ……
}
```

可以将上述程序修改或添加为：

```
#include <iostream>              //输入输出流头文件
#include <fstream>               //文件流头文件
……
main()
{
   ……
   ofstream ofsm("abc.dat");     //创建文件 abc.dat
   ofsm <<…… <<……;              //输出信息至文件 abc.dat
   cout << …… <<……;              //输出信息至显示器
   ……
}
```

这样就可以将运算信息输出到计算机磁盘文件和显示器。在实际程序中，需要输出信息到什么地方随情况灵活使用。

1.9.2 利用命令行命令重定向功能

由于本书编制的示例程序都是在 VC6.0 环境下采用的控制台方式，生成的程序须在 DOS 环境下执行，因此就可以利用其命令行方式的特点，采用它所具有的命令行命令重定向功能，直接生成运算结果文件，将运算结果和提示信息包含在所生成的结果文件中。下面以例子进行说明。

假设，在 E 盘上有第 5 章“矩阵运算”中用于判别矩阵是否是对称正定矩阵的源程序文件 MatrixSymmetryRegular.cpp，对其进行编译连接后，生成可执行文件 chap5.exe，这时就可以使用 DOS 命令，在 E 盘上的 DOS 命令行上键入命令：

```
E:>chap5>msr .dat
```

注意，上行中 E:>后面出现的所有字符表示命令行上的命令，即读者自己在 DOS 命令行上输入的命令，也就是要让 DOS 系统执行的命令操作。DOS 系统只支持 8.3 文件名格式。

在执行上述命令后，DOS 系统就会在 E 盘上自动生成一个新的文件：msr.dat，该文件中包含有运行 chap5.exe 程序后应该产生的所有运行结果，包括生成的数据与提示信息。msr.dat 是一个纯文本文件，即一个常见的 .txt 文件。可以用 DOS 命令 type 打开该文件浏览，即在 DOS 下执行下面的命令：

```
E:>msr.dat
```

在命令执行后，计算机显示器上将显示：

```
matA :
The matrix is not symmetry.

matB :
 K = 1   Determinant = 3
 K = 2   Determinant = 15
 K = 3   Determinant = 13
The matrix is symmetry and regular.

matC :
 K = 1   Determinant = 5
 K = 2   Determinant = 1
 K = 3   Determinant = 2
 K = 4   Determinant = 55
The matrix is regular.
```

这个显示结果与直接运行 chap5.exe 程序后在计算机显示器上显示的结果是一模一样的。另外，在 Windows 环境下显示 msr.dat 的内容会更方便。

像 msr.dat 这样的文件名可以由读者自己随意取，例如，在 Windows 系统下可改为 MatrixDeterminant.txt，因为它的内容就是文本内容。

利用 DOS 命令行命令，也可以将程序的运行结果直接输出至打印机，把运行结果真正打印出来。

若将C++程序移植到其他计算平台的编译环境下，经编制、编译与连接后生成的可执行程序，类似的命令行命令也可以应用，得到相同的效果。

第2章 复数运算

标准C++包含复数模板类 complex 和头文件<complex>，它们是专门为处理复数对象而设计的。包括复数对象的生成，以及针对复数对象的各种重载操作符，极大地满足了不同情况对复数处理的要求。因此，在C++程序的设计中可以直接利用这些已定义好的内容对复数进行所需要的操作，编程人员不必再另外编写有关处理复数的源代码，使编程的效率及可靠性大大提高。

本章首先介绍有关复数的一些基本概念，然后介绍标准C++所定义的复数模板类 complex 和头文件<complex>内容，并通过示例介绍怎样利用这些内容在C++程序中对复数进行各种操作运算，最后给出程序运行的结果及相关说明。

2.1 复数概念

本节简要介绍复数的定义及复数的各种运算。

2.1.1 复数定义

设 $Z=a+ib$，其中，a 和 b 均为实数，$i=\sqrt{-1}$ 时，则称 Z 为复数。a 和 b 分别称为 Z 的实数部分和虚数部分，i 称为虚数单位。

两个复数只有当实数部分和虚数部分都分别相等时才相等。

2.1.2 模与辐角

设 Z 为复数，$|Z|=\sqrt{a^2+b^2}$ 称为复数 Z 的模，常记为 r。

$\arg(Z)=\operatorname{arctg}(b/a)$ 称为复数 Z 的辐角。一个复数有无穷多个辐角，其中一个称为主辐角，记为 $\arg(Z)$，它满足

$$0 \leqslant \arg(Z) < 2\pi$$

并有

$$\arg(Z)=\arg(Z)+2k\pi \qquad (k=0,\pm 1,\pm 2,\cdots)$$

2.1.3 共轭复数

设 $U=a+ib$ 为复数，则 $\overline{U}=a-ib$ 称为 Z 的共轭复数。

2.1.4 虚数单位乘方

$$i=\sqrt{-1} \qquad i^2=-1 \qquad i^3=-i \qquad i^4=1$$

$$i^{4n+1}=i \qquad i^{4n+2}=-1 \qquad i^{4n+3}=-i \qquad i^{4n}=1$$

2.1.5 复数的表示法

1. 坐标表示法

复数 $Z=a+ib$ 可与直角坐标(a,b)建立一一对应关系，无特殊说明，在本书中对复数的描述均采用(a,b)这种表示法。

2. 三角表示法

$$Z=|Z|(\cos\theta+i\sin\theta)=r(\cos\theta+i\sin\theta)$$

3. 指数表示法

$$Z=|Z|e^{i\theta}=re^{i\theta}$$

2.1.6 复数运算

1. 代数式运算公式

(1) 加法

$$(a+ib)+(c+id)=(a+c)+i(b+d)$$

(2) 减法

$$(a+ib)-(c+id)=(a+c)-i(b+d)$$

(3) 乘法

$$(a+ib)*(c+id)=(ac-bd)+i(bc+ad)$$

(4) 除法

$$(a+ib)/(c+id)=((ac+bd)/(c^2+d^2))+i((bc-ad)/(c^2+d^2))$$

2. 三角式运算公式

设 $Z_1=r_1(\cos\theta_1+i\sin\theta_1)$ $Z_2=r_2(\cos\theta_2+i\sin\theta_2)$

则

(1) 乘法

$$Z_1\cdot Z_2=r_1r_2[(\cos\theta_1+\theta_2)+i\sin(\theta_1+\theta_2)]$$

(2) 除法

$$\frac{Z_1}{Z_2}=\frac{r_1}{r_2}[(\cos\theta_1-\theta_2)+i\sin(\theta_1-\theta_2)]$$

(3) 乘幂

$$Z_1^n=r_1^n(\cos n\theta_1+i\sin n\theta_1)$$

其中 n 为正整数。

(4) 开方

$$Z_1^{1/n}=r_1^{1/n}\left(\cos\frac{\theta_1+2k\pi}{n}+i\sin\frac{\theta_1+2k\pi}{n}\right)\quad(k=0,\ 1,\ 2,\ \cdots,\ n-1)$$

3. 指数式运算公式

设 $Z_1=r_1e^{i\theta_1}$ $Z_2=r_2e^{i\theta_2}$

则

(1) 乘法

$$Z_1\cdot Z_2=r_1r_2e^{i(\theta_1+\theta_2)}$$

(2) 除法

$$\frac{Z_1}{Z_2}=\frac{r_1}{r_2}e^{i(\theta_1-\theta_2)}$$

(3) 乘幂

$$Z^n=r^n e^{in\theta}$$

其中 n 为正整数。

(4) 开方

$$Z^{1/n}=r^{1/n}e^{i(\theta+2k\pi)/n}\qquad(k=0,1,2,\cdots,n-1)$$

其中 n 为正整数。

(5) 指数

$$e^z=e^{a+ib}=e^a(\cos b+i\sin b)=e^a\cos b+ie^a\sin b$$

(6) 自然对数

$$\ln Z=\ln(a+ib)=\ln\sqrt{a^2+b^2}+i\operatorname{arctg}\frac{b}{a}$$

(7) 以 10 为底的对数

$$\lg Z=\lg(a+ib)=\lg\sqrt{a^2+b^2}+i\operatorname{arctg}\frac{b}{a}$$

4. 混合运算公式

(1) 正弦

$$\begin{aligned}\sin Z&=\sin(a+ib)=\sin x\cos(ib)+\cos x\sin(ib)\\&=\sin x\left(\frac{e^b+e^{-b}}{2}\right)+i\cos x\left(\frac{e^b-e^{-b}}{2}\right)\end{aligned}$$

(2) 余弦

$$\begin{aligned}\cos Z&=\cos(a+ib)=\cos x\cos(ib)-\sin x\sin(ib)\\&=\cos x\left(\frac{e^b+e^{-b}}{2}\right)-i\sin x\left(\frac{e^b-e^{-b}}{2}\right)\end{aligned}$$

2.2 头文件<complex>及模板类 complex

标准C++定义了头文件<complex>,其中包含复数模板类 complex 及许多针对复数操作符的重载。头文件<complex>及模板类 complex 的详细描述分别见 2.2.1 节和 2.2.2 节,其成员说明见 2.3 节。

2.2.1 头文件<complex>描述

头文件<complex>中包含的成员有:

```
abs · arg · complex · complex<double> · complex<float> · complex<long double> · conj
· cos · cosh · exp · imag · log · log10 · norm · operator! = · operator* · operator+ ·
operator- · operator/ · operator<< · operator== · operator>> · polar · pow · real ·
sin · sinh · sqrt · _STD_COMPLEX
```

头文件<complex>的描述为:

```
namespace std {
#define _STD_COMPLEX
```

```
//  模板类
template<class T>
   class complex;
class complex<float>;
class complex<double>;
class complex<long double>;
//  模板函数
template<class T>
   complex<T> operator + (const complex<T>& lhs, const complex<T>& rhs);
template<class T>
   complex<T> operator + (const complex<T>& lhs, const T& rhs);
template<class T>
   complex<T> operator + (const T& lhs, const complex<T>& rhs);
template<class T>
   complex<T> operator - (const complex<T>& lhs, const complex<T>& rhs);
template<class T>
   complex<T> operator - (const complex<T>& lhs, const T& rhs);
template<class T>
   complex<T> operator - (const T& lhs, const complex<T>& rhs);
template<class T>
   complex<T> operator * (const complex<T>& lhs, const complex<T>& rhs);
template<class T>
   complex<T> operator * (const complex<T>& lhs, const T& rhs);
template<class T>
   complex<T> operator * (const T& lhs, const complex<T>& rhs);
template<class T>
   complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
   complex<T> operator/(const complex<T>& lhs, const T& rhs);
template<class T>
   complex<T> operator/(const T& lhs, const complex<T>& rhs);
template<class T>
   complex<T> operator + (const complex<T>& lhs);
template<class T>
   complex<T> operator - (const complex<T>& lhs);
template<class T>
   bool operator = = (const complex<T>& lhs, const complex<T>& rhs);
template<class T>
   bool operator = = (const complex<T>& lhs, const T& rhs);
template<class T>
   bool operator = = (const T& lhs, const complex<T>& rhs);
template<class T>
   bool operator!  = (const complex<T>& lhs, const complex<T>& rhs);
template<class T>
```

```
    bool operator!  = (const complex<T>& lhs, const T& rhs);
template<class T>
    bool operator!  = (const T& lhs, const complex<T>& rhs);
template<class E, class Ti, class T>
    basic_istream<E, T>& operator>>(basic_istream<E, T>& is,
                                                    complex<T>& x);
template<class E, class T, class U>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os,
                                              const complex<U>& x);
template<class T>
    T real(const complex<T>& x);
template<class T>
    T imag(const complex<T>& x);
template<class T>
    T abs(const complex<T>& x);
template<class T>
    T arg(const complex<T>& x);
template<class T>
    T norm(const complex<T>& x);
template<class T>
    complex<T> conj(const complex<T>& x);
template<class T>
    complex<T> polar(const T& rho, const T& theta = 0);
template<class T>
    complex<T> cos(const complex<T>& x);
template<class T>
    complex<T> cosh(const complex<T>& x);
template<class T>
    complex<T> exp(const complex<T>& x);
template<class T>
    complex<T> log(const complex<T>& x);
template<class T>
    complex<T> log10(const complex<T>& x);
template<class T>
    complex<T> pow(const complex<T>& x, int y);
template<class T>
    complex<T> pow(const complex<T>& x, const T& y);
template<class T>
    complex<T> pow(const complex<T>& x, const complex<T>& y);
template<class T>
    complex<T> pow(const T& x, const complex<T>& y);
template<class T>
    complex<T> sin(const complex<T>& x);
template<class T>
```

```
    complex<T> sinh(const complex<T>& x);
template<class T>
    complex<T> sqrt(const complex<T>& x);
    };
```

在程序中包含头文件<complex>，就可以定义模板类 complex 及其大量的支持模板函数。除非另外说明，否则能够返回复数的该函数返回的虚数部分在半开区间(−π，π]内。

2.2.2 模板类 complex

模板类 complex 的描述为：

```
template<class T>
   class complex {
public:
   typedef T value_type;
   T real() const;
   T imag() const;
   complex(const T& re = 0, const T& im = 0);
   complex(const complex& x);
   complex& operator = (const complex& rhs);
   complex& operator += (const complex& rhs);
   complex& operator -= (const complex& rhs);
   complex& operator *= (const complex& rhs);
   complex& operator/= (const complex& rhs);
   complex& operator = (const complex& rhs);
   complex& operator = (const T& rhs);
   complex& operator += (const T& rhs);
   complex& operator -= (const T& rhs);
   complex& operator *= (const T& rhs);
   complex& operator/= (const T& rhs);
   friend complex<T>
      operator + (const complex<T>& lhs, const T& rhs);
   friend complex<T>
      operator + (const T& lhs, const complex<T>& rhs);
   friend complex<T>
      operator - (const complex<T>& lhs, const T& rhs);
   friend complex<T>
      operator - (const T& lhs, const complex<T>& rhs);
   friend complex<T>
      operator * (const complex<T>& lhs, const T& rhs);
   friend complex<T>
      operator * (const T& lhs, const complex<T>& rhs);
   friend complex<T>
      operator/(const complex<T>& lhs, const T& rhs);
   friend complex<T>
```

```
    operator/(const T& lhs, const complex<T>& rhs);
  friend bool operator = = (const complex<T>& lhs, const T& rhs);
  friend bool operator = = (const T& lhs, const complex<T>& rhs);
  friend bool operator! = (const complex<T>& lhs, const T& rhs);
  friend bool operator! = (const T& lhs, const complex<T>& rhs);
  };
```

该模板类描述的对象本身又包含两个 T 类型的对象，其中一个描述复数的实数部分，另一个描述虚数部分。类型为 T 的类对象具有如下特点。

- 有一个 public 类型的默认构造函数、析构函数和赋值操作符，各函数的行为为常规操作。
- 可以赋整数或浮点类型的值，也可以强制转换为上述类型的值，其行为为常规操作。
- 可以专为浮点类型的常规行为定义算术操作符，其行为为常规操作。

模板类 complex 的显式定制有三个浮点类型。在 complex 的当前实现中，任何其他类型为 T 的值在实际计算中将转换为进行算术运算的 double 类型，double 类型的结果值将赋回原所包含的 T 类型的对象。

2.3 模板类 complex 与<complex>成员

下面给出<complex>和模板类 complex 的构成成员，其具体应用参阅 2.4 节中示例。

2.3.1 类型定义

```
complex::value_type
typedef T value_type;
```

该类型是模板参数 *T* 的同义词。

2.3.2 模板类 complex<float>

模板类 complex<float>的描述为：

```
class complex<float> {
public:
   complex(float re = 0, float im = 0);
   explicit complex(const complex<double>& x);
   explicit complex(const complex<long double>& x);
// rest same as template class complex
   };
```

该显式指定的模板类描述的对象存储两个 float 类型的对象，其中一个描述复数的实数部分，另一个描述虚数部分。该显式描述的模板类仅仅定义的构造函数不同。其中第一个构造函数初始化存储在 *re* 的实数部分和存储在 im 的虚数部分；其他两个构造函数初始化存储在 x.real()的实数部分和存储在 x.imag()的虚数部分。

2.3.3 模板类 complex<double>

模板类 complex<double>的描述为：

```
class complex<double> {
public:
   complex(double re = 0, double im = 0);
   explicit complex(const complex<float>& x);
   explicit complex(const complex<long double>& x);
// rest same as template class complex
   };
```

该显式指定的模板类描述的对象存储两个 double 类型的对象，其中一个描述复数的实数部分，另一个描述虚数部分。该显式描述的模板类仅仅定义的构造函数不同。其中第一个构造函数初始化存储在 re 的实数部分和存储在 im 的虚数部分；其他两个构造函数初始化存储在 x. real()的实数部分和存储在 x. imag()的虚数部分。

2.3.4 模板类 complex<long double>

模板类 complex<long double>的描述为：

```
class complex<long double> {
public:
   complex(long double re = 0, long double im = 0);
   explicit complex(const complex<float>& x);
   explicit complex(const complex<long double>& x);
   // rest same as template class complex
   };
```

该显式指定的模板类描述的对象存储两个 long double 类型的对象，其中一个描述复数的实数部分，另一个描述虚数部分。该显式描述的模板类仅仅定义的构造函数不同。其中第一个构造函数初始化存储在 re 的实数部分和存储在 im 的虚数部分；其他两个构造函数初始化存储在 x. real()的实数部分和存储在 x. imag()的虚数部分。

2.3.5 构造函数

```
complex::complex
complex(const T& re = 0, const T& im = 0);
complex(const complex& x);
```

第一个构造函数初始化存储在 re 的实数部分和存储在 im 的虚数部分；第二个构造函数用存储在 x. real()的实数部分和存储在 x. imag()的虚数部分进行初始化。

如果翻译器不支持成员模板函数，则模板

```
template<class U>
   complex(const complex<U>& x);
```

替换为

```
complex(const complex& x);
```

这是一个备份构造函数。

2.3.6 取实数部分

```
complex::real
T real() const;
```

该成员函数返回存储的实数部分。

2.3.7 取虚数部分

```
complex::imag
T imag() const;
```

该成员函数返回存储的虚数部分。

2.3.8 复数实数部分

```
real
template<class T>
T real(const complex<T>& x);
```

该函数返回 x 的实数部分。

2.3.9 复数虚数部分

```
imag
template<class T>
T imag(const complex<T>& x);
```

该函数返回 x 的虚数部分。

2.3.10 赋值

```
complex:: operator =
complex& operator = (const complex& rhs);
complex& operator = (const T& rhs);
```

第一个成员函数将存储的实数部分替换为 rhs.real(),虚数部分替换为 rhs.imag(),然后返回 * this。

第二个成员函数将存储的实数部分替换为 rhs,虚数部分替换为 0,然后返回 * this。

如果翻译器不支持成员模板函数,则模板

```
template<class U>
  complex& operator = (const complex<U>& rhs);
```

替换为

```
complex& operator = (const complex& rhs);
```

这是默认的赋值操作符。

2.3.11 赋值加

```
complex:: operator + =
complex& operator + = (const complex& rhs);
complex& operator + = (const T& rhs);
```

第一个成员函数将存储的实数部分和虚数部分替换为 * this 和 rhs 的复数之和的对应部分,然后返回 * this。

第二个成员函数将存储的实数部分和虚数部分分别加上 rhs,然后返回 * this。

如果翻译器不支持成员模板函数,则模板

```
template<class U>
   complex& operator += (const complex<U>& rhs);
```

替换为

```
complex& operator += (const complex& rhs);
```

2.3.12 赋值减

```
complex::operator -=
complex& operator -= (const complex& rhs);
complex& operator -= (const T& rhs);
```

第一个成员函数将存储的实数部分和虚数部分替换为 * this 和 rhs 的复数之差的对应部分，然后返回 * this。

第二个成员函数将存储的实数部分和虚数部分分别减去 rhs，然后返回 * this。

如果翻译器不支持成员模板函数，则模板

```
template<class U>
   complex& operator -= (const complex<U>& rhs);
```

替换为

```
complex& operator -= (const complex& rhs);
```

2.3.13 赋值乘

```
complex::operator *=
complex& operator *= (const complex& rhs);
complex& operator *= (const T& rhs);
```

第一个成员函数将存储的实数部分和虚数部分替换为 * this 和 rhs 的复数乘积的对应部分，然后返回 * this。

第二个成员函数将存储的实数部分和虚数部分分别乘以 rhs，然后返回 * this。

如果翻译器不支持成员模板函数，则模板

```
template<class U>
   complex& operator *= (const complex<U>& rhs);
```

替换为

```
complex& operator *= (const complex& rhs);
```

2.3.14 赋值除

```
complex::operator/=
complex& operator/= (const complex& rhs);
complex& operator/= (const T& rhs);
```

第一个成员函数将存储的实数部分和虚数部分替换为 * this 和 rhs 的复数之商的对应部分，然后返回 * this。

第二个成员函数将存储的实数部分和虚数部分分别乘以 1/rhs，然后返回 * this。

如果翻译器不支持成员模板函数，则模板

```
template<class U>
   complex& operator/= (const complex<U>& rhs);
```

替换为

```
complex& operator/ = (const complex& rhs);
```

2.3.15 加法

```
operator +
template<class T>
    complex<T> operator + (const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator + (const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator + (const T& lhs, const complex<T>& rhs);
```

上述每个操作符将两个操作数的类型转换为返回类型，然后返回转换后的 lhs 和 rhs 的复数之和。

最后一个一元操作符返回 lhs。

2.3.16 减法

```
operator -
template<class T>
    complex<T> operator - (const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator - (const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator - (const T& lhs, const complex<T>& rhs);
```

上述每个操作符将两个操作数的类型转换为返回类型，然后返回转换后的 lhs 和 rhs 的复数之差。

最后一个一元操作符返回的值，实数部分为－real(lhs)，虚数部分为－imag(lhs)。

2.3.17 乘法

```
operator *
template<class T>
    complex<T> operator * (const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator * (const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator * (const T& lhs, const complex<T>& rhs);
```

上述每个操作符将两个操作数的类型转换为返回类型，然后返回转换后的 lhs 和 rhs 的复数乘积。

2.3.18 除法

```
operator/
template<class T>
    complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator/(const complex<T>& lhs, const T& rhs);
```

```
template<class T>
   complex<T> operator/(const T& lhs, const complex<T>& rhs);
```

上述每个操作符将两个操作数的类型转换为返回类型，然后返回转换后的 lhs 除以 rhs 的复数的商。

2.3.19 复数绝对值

```
abs
template<class T>
T abs(const complex<T>& x);
```

该函数返回 x 的绝对值。

2.3.20 复数辐角

```
arg
template<class T>
T arg(const complex<T>& x);
```

该函数返回 x 的辐角。

2.3.21 共轭复数

```
conj
template<class T>
complex<T> conj(const complex<T>& x);
```

该函数返回 x 的共轭。

2.3.22 正弦

```
sin
template<class T>
T sin(const complex<T>& x);
```

该函数返回 x 的正弦。

2.3.23 余弦

```
cos
template<class T>
T cos(const complex<T>& x);
```

该函数返回 x 的余弦。

2.3.24 双曲正弦

```
sinh
template<class T>
T sinh(const complex<T>& x);
```

该函数返回 x 的双曲正弦。

2.3.25 双曲余弦

```
cosh
```

```
template<class T>
T cosh(const complex<T>& x);
```

该函数返回 x 的双曲余弦。

2.3.26 指数

```
exp
template<class T>
T exp(const complex<T>& x);
```

该函数返回 x 的指数。

2.3.27 自然对数

```
log
template<class T>
T log(const complex<T>& x);
```

该函数返回 x 的自然对数。x 不包括负实轴。

2.3.28 以 10 为底的复数对数

```
log10
template<class T>
T log10(const complex<T>& x);
```

该函数返回 x 的以 10 为底的对数。x 不包括负实轴。

2.3.29 平方

```
norm
template<class T>
T norm(const complex<T>& x);
```

该函数返回 x 的平方。

2.3.30 复数极坐标值

```
polar
template<class T>
   complex<T> polar(const T& rho, const T& theta = 0);
```

该函数返回的复数值,其长度(模)为 *rho*,辐角为 *theta*。

2.3.31 乘幂

```
pow
template<class T>
   complex<T> pow(const complex<T>& x, int y);
template<class T>
   complex<T> pow(const complex<T>& x, const T& y);
template<class T>
   complex<T> pow(const complex<T>& x, const complex<T>& y);
```

```
template<class T>
    complex<T> pow(const T& x, const complex<T>& y);
```

上述每个函数均将两个操作符的类型转换为返回类型，然后返回转换后的 x 的 y 次方。x 不包括负实轴。

2.3.32 平方根

```
sqrt
template<class T>
T sqrt(const complex<T>& x);
```

该函数返回 x 的平方根，这里 x 的辐角在半开区间$(-\pi, \pi]$内。

2.3.33 判断两复数相等

```
operator==
template<class T>
bool operator==(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
bool operator==(const complex<T>& lhs, const T& rhs);
template<class T>
bool operator==(const T& lhs, const complex<T>& rhs);
```

仅当 real(lhs)==real(rhs)||imag(lhs)==imag(rhs)时，上述每个操作符返回 true。

2.3.34 判断两复数不相等

```
operator!=
template<class T>
    bool operator!=(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    bool operator!=(const complex<T>& lhs, const T& rhs);
template<class T>
    bool operator!=(const T& lhs, const complex<T>& rhs);
```

仅当 real(lhs)!=real(rhs)||imag(lhs)!=imag(rhs)时，上述每个操作符返回 true。

2.3.35 复数输入

```
operator<<
template<class E, class T, class U>
basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os,
        const complex<U>& x);
```

该模板函数通过执行下述语句将复数值插入到输出流 os 中：

```
basic_ostringstream<E, T> ostr;
ostr.flags(os.flags());
ostr.imbue(os.imbue());
ostr.precision(os.precision());
ostr << '(' << real(x) << ', '<< imag(x) << ')';
```

```
os << ostr.str().c_str();
```

因此，如果 os.width()大于 0，则可在数值的括号对之前或之后插入，括号对内的数值不包含插入的内容。该函数返回 os。

2.3.36 复数输出

```
operator>>
template<class E, class T, class U>
basic_ostream<E, T>& operator>>(basic_ostream<E, T>& os,
        const complex<U>& x);
```

该模板函数通过执行下述语句从输入流 is 中析取一个复数值：

```
is >> ch && ch == '('
is >> re >> ch && ch == ','
is >> re >> ch && ch == ')'
```

这里，ch 为 E 类型对象，re 和 im 为 U 类型对象。

如果结果为 true，则函数将 re 存储在 x 的实数部分，将 im 存储在 x 的虚数部分。不管怎样，函数均返回 is。

2.3.37 宏_STD_COMPLEX

```
_STD_COMPLEX
#define _STD_COMPLEX
```

该宏有一个未指明的扩展，用于指示根据头文件的说明进行操作。

2.4 复数运算示例

2.4.1 示例

示例给出了利用复数模板类 complex 提供的方法(函数)所得到的结果，包括定义复数对象，取其复数、实数部分和虚数部分，对复数的加、减、乘、除、相等和不相等运算，以及对复数的取模，求共轭、正弦、余数、对数、指数和平方根等各种运算。

示例程序如下：

```
// 示例 ComplexExample.cpp

#include <iostream>          //输入输出流头文件
#include <complex>           //复数类头函数
using namespace std;         //名字空间

void main()          // 定义控制台应用程序的入口点
{
  double dcr = -3.1, dci = 14.5;              //定义两个 double 变量
  complex<double> cda(dcr,dci);               //定义复数对象 cda
  cout << " 1 cda = " << cda << endl;
  complex<double> cdb(cda);                   //定义复数对象 cdb
```

```
    cout << " 2 cdb = " << cdb << endl;
    complex<double> cdc( - 2.4, 7.9);          //定义复数对象 cdc
    cout << " 3 cdc = " << cdc << endl;
    complex<int> cia(5,8);                     //定义复数对象 cia
    complex<float> cla;                        //定义复数对象 cla
    cout << " 4 cia = " << cia << " cla = " << cla << endl;
    cia = 9;                                   //cia 实、虚部分别赋 9 和 0
    cout << " 5 cia = " << cia << endl;
    dcr = cdc.real();                          //取 cdc 实数部分
    cout << " 6 cda.real() = " << dcr << endl;
    dci = cdc.imag();                          //取 cdc 虚数部分
    cout << " 7 cda.imag() = " << dci << endl;
    cout << " 8 real(cdb) = " << real(cdb) << endl;
    cout << " 9 imag(cdb) = " << imag(cdb) << endl;
    cout << "10 (cdc = cda) = " << (cdc = cda) << endl;
    cout << "11 (cda + = cdb) = " << (cda + = cdb) << endl;
    cout << "12 (cda - = cdb) = " << (cda - = cdb) << endl;
    cout << "13 (cdb * = cdc) = " << (cdb * = cdc) << endl;
    cout << "14 (cda / = cdc) = " << (cda / = cdc) << endl;
    cout << "15 (cda = cdc + cdb) = " << (cda = cdc + cdb) << endl;
    cout << "16 (cda = cda - cdb) = " << (cda = cda - cdb) << endl;
    cout << "17 (cda = cda * cdb) = " << (cda = cda * cdb) << endl;
    cout << "18 (cda = cda / cdb) = " << (cda = cda / cdb) << endl;
    cout << "19 abs(cda) = " << abs(cda) << endl;
    cout << "20 arg(cda) = " << arg(cda) << endl;
    cout << "21 conj(cna) = " << (conj(cda)) << endl;
    cout << "22 sin(cda) = " << sin(cda) << endl;
    cout << "23 cos(cda) = " << cos(cda) << endl;
    cout << "24 sinh(cda) = " << sinh(cda) << endl;
    cout << "25 cosh(cda) = " << cosh(cda) << endl;
    cout << "26 exp(cda) = " << exp(cda) << endl;
    cout << "27 log(cda) = " << log(cda) << endl;
    cout << "28 log10(cda) = " << log10(cda) << endl;
    cout << "29 norm(cda) = " << norm(cda) << endl;
    cout << "30 polar(dcr, dci) = " << polar(dcr, dci) << endl;
    cout << "31 pow(cdc, 3) = " << pow(cdc, 3) << endl;
    cout << "32 sqrt(cdc) = " << sqrt(cdc) << endl;
    cout << "33 (cdc = = cdb) = " << (cdc = = cdb) << endl;
    cout << "34 (cdc ! = cdb) = " << (cdc! = cdb) << endl;
    cout << "35 pow(cdb, cda) = " << pow(cdb, cda) << endl;
}
```

2.4.2 示例运行结果

```
1 cda = ( - 3.1,14.5)
```

```
2 cdb = (-3.1,14.5)
3 cdc = (-2.4,7.9)
4 cia = (5,8)          cla = (0,0)
5 cia = (9,0)
6 cda.real() = -2.4
7 cda.imag() = 7.9
8 real(cdb) = -3.1
9 imag(cdb) = 14.5
10 (cdc = cda) = (-3.1,14.5)
11 (cda += cdb) = (-6.2,29)
12 (cda -= cdb) = (-3.1,14.5)
13 (cdb *= cdc) = (-200.64,-89.9)
14 (cda /= cdc) = (1,0)
15 (cda = cdc + cdb) = (-203.74,-75.4)
16 (cda = cda - cdb) = (-3.1,14.5)
17 (cda = cda * cdb) = (1925.53,-2630.59)
18 (cda = cda / cdb) = (-3.1,14.5)
19 abs(cda) = 14.8277
20 arg(cda) = 1.78142
21 conj(cna) = (-3.1,-14.5)
22 sin(cda) = (-41222.2,-990522)
23 cos(cda) = (-990522,41222.2)
24 sinh(cda) = (3.9313,10.3974)
25 cosh(cda) = (-3.94729,-10.3553)
26 exp(cda) = (-0.0159891,0.0421163)
27 log(cda) = (2.6965,1.78142)
28 log10(cda) = (1.17107,0.77366)
29 norm(cda) = 219.86
30 polar(dcr, dci) = (0.110405,-2.39746)
31 pow(cdc, 3) = (1925.53,-2630.59)
32 sqrt(cdc) = (2.42154,2.99397)
33 (cdc == cdb) = 0
34 (cdc != cdb) = 1
35 pow(cdb, cda) = (1.74564e+009,-7.20662e+009)
```

2.4.3 示例说明

在示例程序中定义的 double 变量 *dcr* 与 *dci* 分别作为复数的实数部分与虚数部分使用。

程序中定义了三个 double 类型的复数对象 cda、cdb 和 cdc，第一个和第二个对象的初始化使用了第一种构造函数，第二个对象的初始化使用了第二种构造函数，并分别用输出语句 1、2、3 输出其值。程序中还分别定义了一个 int 类型和一个 float 类型的复数对象，用输出语句 4 输出其初始值。C++中不能对复数型变量直接用常量赋值，例如：

```
complex<int> cia(14,8);          //定义 int 型复变量 cia
cia = (2, 7);                    //错误
```

```
cia = complex<int>(2, 7);          //可以
```

输出语句5～34分别输出了对复数的各种操作结果，其输出语句的编号与2.3节中所列各成员函数的编号是对应的，可相互对照参阅。输出语句35是pow函数的另一种参数使用法。

第3章　随机数生成

随机数被广泛地应用在各种计算机数值计算的算法中，特别是用在各种数值计算迭代算法的初始值取值上，以及各种统计计算的随机算法取值上。本章介绍随机数生成的各种数值算法，它们产生的随机数大都是伪随机数，但已经能够很好地满足实际计算的需要。每种算法都用C++程序以函数的形式实现。除第一个 rand()函数为C++标准函数，可以直接调用外，其他几种都定义了相应的函数，详细介绍参阅本章各节。

本章 3.1 节介绍了随机数生成的各种算法及头文件 random.h。random.h 包括所涉及到的宏及随机函数的原型，另外在本书配套的光盘中还有一个包含了随机函数定义的相关头文件 random.inl。本章 3.2 节给出了各个随机函数的详细介绍，包括其使用的示例及相关说明。

3.1　随机数生成与头文件

3.1.1　随机数生成

本章介绍的随机数生成算法及程序包括：

- 产生一个介于 0～32767 之间的随机整数；
- 产生一个[0,1]区间内均匀分布伪随机数；
- 产生多个[0,1]区间内均匀分布伪随机数；
- 产生任意[a,b]区间内一个均匀分布伪随机整数；
- 产生任意[a,b]区间内均匀分布伪随机整数序列；
- 产生一个任意均值与方差的正态分布随机数；
- 产生任意均值与方差的正态分布随机数序列。

3.1.2　random.h

```
//random.h          宏定义及随机函数(方法)原型头文件

#ifndef _RANDOM_H
#define _RANDOM_H

#define RandCoef2053 2053
#define RandCoef13849 13849
#define MODUL65536 65536

#include <comm.h>     //公共部分头文件
//产生一个[0,1]区间内均匀分布伪随机数
inline double rand_01_One(double& seed);
```

```
//产生多个[0,1]区间内均匀分布伪随机数
inline void
rand_01_Series(double& seed, valarray<double>& dp, const size_t stCount);

//产生任意[a,b]区间内一个均匀分布伪随机整数
inline size_t
rand_ab_One(size_t a, size_t b, size_t& seed);

//产生任意[a,b]区间内均匀分布伪随机整数序列
inline void
rand_ab_Series(size_t a, size_t b, size_t& seed, valarray<size_t>& sp, size_t stCount);

//产生一个任意均值与方差的正态分布随机数
inline double
rand_NormalDistributing_One(double mu, double ro, double& seed);

//产生任意均值与方差的正态分布随机数序列
inline void
rand_NormalDistributing_Series(double mu, double ro, double seed, valarray<double>& dp, size_t
stCount);

#include <random.inl>

#endif //_RANDOM_H
```

3.2 随机数生成算法

3.2.1 产生一个介于0～32767之间的随机整数

1. 功能

产生一个介于0到RAND_MAX之间的随机整数，RAND_MAX为int类型的最大正整数(通常为32767)。

2. 函数原型

```
#include <stdlib.h>        //或#include <cstdlib>
int rand<void>
```

3. 参数

无。

4. 返回值

整型随机数值。

5. 示例

连续生成10个随时间变化的在0～32767之间均匀分布的伪随机数。

主函数及示例程序如下：

```
// rand.cpp        C++提供的rand()函数
// 该程序利用time(与srand)函数作为随机数生成器的种子,生成10个随机整数,然后输出

#include <random.h>   //随机算法头文件
#include <iostream>   //输入输出流头文件
#include <time.h>     //时间头文件
using namespace std;  //名字空间

void main( void )
{
  // 随机数生成器需要的种子值使用当前时间,以便每次程序运行时该种子值都不相同
  srand( (size_t)time( NULL ) );

  // 输出10个随机数
  for (size_t i = 0; i < 10; i++){
      cout.width(6);
      cout << rand() << endl;
  }
}
```

6. 示例结果

```
16001
18413
31756
  7123
20066
26391
  279
17426
  1114
16518
```

7. 说明

rand()函数返回随机数序列中的下一个数(实际上是一个伪随机数序列,序列中的每一个数是由对其前面的数字进行复杂变换得到的)。为了模仿真正的随机性,首先要调用srand()函数给序列设置一个种子。为了更好地满足随机性,还使用了时间函数time(),以便取到一个随时间变化的值,使每次运行rand()函数时从srand()函数所得到的种子值不相同。

可以用RAND_MAX来除所得到的结果,以获得一个介于0和1.0之间的浮点随机数,然后可通过倍乘和取整来把这些数转换成整数。例如,以下的代码就以相等的概率返回整数0、1、2、3和4:

```
include <stdlib.h>
//…
cout << floor(5 * rand() /RAND_MAX) << endl;
```

其中,floor函数进行向下取整。

rand()函数是C++的标准函数，可以在C++环境下直接调用。该函数支持 ANSI C 和 UNIX C，UNIX 包括 XENIX 及其他类 UNIX 系统，也包括 Windows 下 POSIX 子系统。

3.2.2 产生一个[0,1]区间内均匀分布伪随机数

1. 功能

产生一个[0,1]区间内均匀分布伪随机数。

2. 算法

使用如下公式产生一个[0,1]区间内均匀分布的伪随机数：

$$\begin{cases} r_k = \mathrm{mod}(2053\ r_{k-1} + 13849,\ M) & k = 1,2,\cdots \\ p_k = r_k / M \end{cases}$$

其中 $M=2^{16}$。

3. 函数原型

```
//产生一个[0,1]区间内均匀分布伪随机数
inline double rand_01_One(double& seed);
```

4. 参数

seed：类型为 double 的引用，传送随机数种子。

5. 返回值

double 型随机数值。

6. 函数说明

为了更好地满足随机性，使用了时间函数 time()，以便取到一个随时间变化的值，使每次运行 rand_01_One()时所得到的种子值不相同。

7. 示例

连续生成 10 个随时间变化的、在[0,1]区间内均匀分布的伪随机数。

主函数及示例程序如下：

```
//rand_01_One.cpp        分别产生 10 个[0, 1]区间内均匀分布的伪随机数

#include <random.h>   //随机算法头文件
#include <iostream>   //输入输出流头文件
#include <time.h>     //时间头文件
using namespace std;  //名字空间
void main(void)
{
  double seed = (double)time(NULL);  //取种子
  cout.setf(ios::fixed|ios::left);   //输出数据为定点法，靠左对齐
  cout.precision(6);                 //精度 6 位
  for(size_t st = 0; st < 10; st++)
  {
      double rand = rand_01_One(seed);  //生成[0,1]区间内的随机数
      cout << " random = " << rand << endl;
  }
}
```

8. 示例结果

```
random = 0.571930
random = 0.383469
random = 0.472412
random = 0.073380
random = 0.859467
random = 0.696152
random = 0.410828
random = 0.640457
random = 0.069855
random = 0.623093
```

3.2.3 产生多个[0,1]区间内均匀分布伪随机数

1. 功能

产生[0,1]区间内均匀分布的伪随机数序列。

2. 算法

使用如下公式产生一个[0,1]区间内均匀分布的伪随机数：

$$\begin{cases} r_k = \mathrm{mod}(2053\ r_{k-1} + 13849,\ M) & k = 1,2,\cdots \\ p_k = r_k / M \end{cases}$$

其中 $M = 2^{16}$。

3. 函数原型

```
//产生多个[0,1]区间内均匀分布伪随机数
inline void
rand_01_Series(double& seed, valarray<double>& dp, const size_t stCount);
```

4. 参数

seed：类型为 double 的引用，传送随机数种子。

dp：类型为 valarray<double>的引用，存放生成的随机数。

srCount：类型为非负整数，指定要生成的随机数个数。

5. 返回值

无。

6. 函数说明

为了更好地满足随机性，使用了时间函数 time()，以便取到一个随时间变化的值，使每次运行 rand_01_Series()时所得到的种子值不相同。

7. 示例

连续生成 40 个随时间变化的、在[0,1]区间内均匀分布的伪随机数。

主函数及示例程序如下：

```
//rand_01_Series.cpp 产生[0,1]区间内均匀分布伪随机数序列

#include <random.h>   //随机算法头文件
#include <iostream>   //输入输出流头文件
#include <time.h>     //时间头文件
using namespace std;  //名字空间
```

```
void main(void)
{
  double seed = (double)time(NULL);  //取种子
  valarray<double> dp(40);
  size_t stNum = 40;
  cout.setf(ios::fixed|ios::left);   //输出数据为定点法,靠左对齐
  cout.precision(6);                 //精度 6 位
  rand_01_Series(seed, dp, stNum);   //生成[0,1]区间 40 个均匀随机数
  ValarrayPrint(dp);                 //结果输出至显示器
  cout << endl;
}
```

8. 示例结果

0.655350	0.644318	0.995407	0.782104	0.871841
0.101776	0.157700	0.968567	0.679153	0.513336
0.090500	0.007568	0.749161	0.238373	0.590683
0.883484	0.003738	0.886261	0.705124	0.830688
0.614761	0.316376	0.730698	0.333557	0.004105
0.638092	0.214279	0.126465	0.843643	0.210785
0.952744	0.193787	0.055252	0.643829	0.992966
0.769897	0.810806	0.796600	0.631821	0.339172

3.2.4 产生任意[a,b]区间内一个均匀分布伪随机整数

1. 功能

产生任意[a,b]区间内一个均匀分布伪随机整数。

2. 算法

首先,产生在区间[0,s]内均匀分布的随机整数,其计算公式如下:

$$\begin{cases} r_j = \mathrm{mod}(5r_{j-1}, 4M) & j = 1,2,\cdots \\ p_j = r_j/4 \end{cases}$$

其中,初值为 $r_0 \geqslant 1$ 的奇数(随机整数种子),$s=b-a+1$,$M = 2^k$,$k=[\log_2 s]+1$。

然后将每个随机整数加上 a,即得到实际需要的随机整数值。

3. 函数原型

```
//产生任意[a,b]区间内一个均匀分布伪随机整数
inline size_t
rand_ab_One(size_t a, size_t b, size_t& seed);
```

4. 参数

a:非负整数类型,产生随机数区间左端点。

b:非负整数类型,产生随机数区间右端点。

seed:类型为非负整数的引用,传送随机数种子。

5. 返回值

非负整数的随机数值。

6. 函数说明

为了更好地满足随机性,使用了时间函数 time(),以便取到一个随时间变化的值,使每次

运行 rand_ab_One()时所得到的种子值不相同。

7. 示例

生成 40 个随时间变化的、在[1000,9999]区间内均匀分布的伪随机数。

主函数及示例程序如下：

```
//rand_ab_One.cpp     产生任意[a,b]区间内一个均匀分布伪随机整数

#include <random.h>   //随机算法头文件
#include <iostream>   //输入输出流头文件
#include <time.h>     //时间头文件
using namespace std;  //名字空间

//从[1000,9999]中产生 40 个随机整数
void main(void)
{
  size_t a = 1000, b = 9999;
  size_t seed = time(NULL);          //取种子
  for(size_t st = 0; st < 40; st++)
  {
      if (st % 5 == 0) cout << endl;
      cout << rand_ab_One(a, b, seed) << "\t";
  }
  cout << endl;
}
```

8. 示例结果

```
4240   1469   3346   9230   9383
8390   5183   5532   7277   8505
5758   8407   5268   5957   9402
2705   9526   1164   1821   5106
5147   5352   6377   4359   1412
3061   3379   3494   1485   3426
9369   5285   6042   9827   8689
6678   6269   1659   4296   1097
```

3.2.5 产生任意[a,b]区间内均匀分布伪随机整数序列

1. 功能

产生任意[a,b]区间内均匀分布伪随机整数序列。

2. 算法

首先，产生在区间[0,s]内均匀分布的随机整数，其计算公式如下：

$$\begin{cases} r_j = \mathrm{mod}(5r_{j-1}, 4M) & j = 1,2,\cdots \\ p_j = r_j/4 \end{cases}$$

其中，初值为 $r_0 \geqslant 1$ 的奇数(随机整数种子)，$s=b-a+1$，$M = 2^k$，$k=[\log_2 s]+1$。

然后将每个随机整数加上 a，即得到实际需要的随机整数值。

3. 函数原型

```
//产生任意[a,b]区间内均匀分布伪随机整数序列
inline void
rand_ab_Series(size_t a, size_t b, size_t& seed, valarray<size_t>& sp, size_t stCount);
```

4. 参数

a:非负整数类型,产生随机数区间左端点。

b:非负整数类型,产生随机数区间右端点。

seed:类型为非负整数的引用,传送随机数种子。

sp:类型为 valarray<size_t>的引用,存放生成的随机数序列。

stCount:类型为非负整数,指定要生成随机数个数。

5. 返回值

无。

6. 函数说明

为了更好地满足随机性,使用了时间函数 time(),以便取到一个随时间变化的值,使每次运行 rand_ab_Series()时所得到的种子值不相同。

7. 示例

生成 40 个随时间变化的、在[1000,9999]区间内均匀分布的伪随机数。

主函数及示例程序如下:

```
//rand_ab_Series.cpp 产生任意[a,b]区间内均匀分布伪随机整数序列

#include <random.h>   //随机算法头文件
#include <iostream>   //输入输出流头文件
#include <time.h>     //时间头文件
using namespace std;  //名字空间

//从[1000,9999]中产生 40 个随机整数
void main(void)
{
  size_t a = 1000, b = 9999, stCount = 40;
  valarray<size_t> rip(40);
  size_t seed = time (NULL);                  //取种子
  rand_ab_Series(a, b, seed, rip, stCount); //生成随机数
  ValarrayPrint(rip);                         //输出至显示器
  cout << endl;
}
```

8. 示例结果

```
7290  5085  5042  4827  3752
4270  3935  8845  7458  5465
6942  2100  6501  7459  5590
7567  1068  1341  2706  9531
1289  2446  8231  4388  1557
3786  5120  5217  5702  8127
```

```
3868  7170  7800  2233  7166
7700  1733  4666  2947  5615
```

3.2.6 产生一个任意均值与方差的正态分布随机数

1. 功能

产生一个任意均值与方差的正态分布随机数。

2. 算法

产生均值为 μ,方差为 σ^2 的正态分布随机数 y 的计算公式:

$$y=\mu+\sigma\frac{(\sum_{k=0}^{n-1}rnd_k)-\frac{n}{2}}{\sqrt{n/12}}$$

其中 n 足够大。通常取 12 时,其近似程序是相当好了,此时有

$$y=\mu+\sigma(\sum_{k=0}^{11}rnd_k)-6$$

其中 rnd_k 为 0 到 1 之间均匀分布的随机数。

3. 函数原型

```
//产生一个任意均值与方差的正态分布随机数
inline double
rand_NormalDistributing_One(double mu, double ro, double& seed);
```

4. 参数

mu:double 类型,正态分布的均值 μ。

ro:double 类型,正态分布的方差 $\sigma^2=ro^2$ 值。

seed:类型为 double 的引用,传送随机数种子初值,再返回一个新值。

5. 返回值

类型为 double 的随机数值。

6. 函数说明

rand_NormalDistributing_One()函数的参数为 mu 和 ro(μ 和 σ 值),要事先给定,也可从其他程序中获得。

如果算法中的 n 值要取得更大(超过 12),rand_NormalDistributing_One()程序可根据正态分布随机数 y 的计算公式稍加修改。

7. 示例

分别取 μ 值和 σ 值为 1.0 和 1.5,生成 40 个正态分布的随机数。

主函数及示例程序如下:

```
//rand_NormalDistributing_One.cpp  产生任意均值与方差的一个正态分布随机数

#include <random.h>   //随机算法头文件
#include <iostream>   //输入输出流头文件
#include <time.h>     //时间头文件
using namespace std;  //名字空间

void main(void)
```

```
{
  double mu = 1.0, ro = 1.5;
  double seed = (double)time(NULL);  //取种子
  cout.setf(ios::fixed|ios::left);   //输出数据为定点法,靠左对齐
  cout.precision(6);                 //精度6位
  for(size_t st = 0; st < 40; st++)
  {
      if (st % 5 == 0) cout << endl;
      cout << rand_NormalDistributing_One(mu, ro, seed) << "\t";
  }
  cout << endl;
}
```

8. 示例结果

```
-0.531540    0.047440    3.632278   -2.808273   -0.805466
 2.609451   -1.094772    0.550613    0.514359    2.265213
 0.271927    2.503250    1.927933    0.514725    3.232376
-1.450363    0.435257    0.357986    1.786575    0.689774
-0.963669   -1.205002    0.434525    1.423660    2.231155
 1.825760    0.676224   -0.748703    4.019730   -1.049728
-0.488327    0.172684    1.402054   -0.831467    1.440872
 1.187820    1.878128   -0.519455   -1.036179    0.796707
```

3.2.7 产生任意均值与方差的正态分布随机数序列

1. 功能

产生任意均值与方差的正态分布随机数序列。

2. 算法

产生均值为 μ、方差为 σ^2 的正态分布随机数 y 的计算公式：

$$y=\mu+\sigma\frac{\left(\sum_{k=0}^{n-1}rnd_k\right)-\frac{n}{2}}{\sqrt{n/12}}$$

其中 n 足够大。通常取 12 时，其近似程序就相当好了，此时有

$$y=\mu+\sigma\left(\sum_{k=0}^{11}rnd_k\right)-6$$

其中 rnd_k 为 0 到 1 之间均匀分布的随机数。

3. 函数原型

```
//产生任意均值与方差的正态分布随机数序列
inline void
rand_NormalDistributing_Series(double mu, double ro, double seed,
                               valarray<double>& dp, size_t stCount);
```

4. 参数

mu：double 类型，正态分布的均值 μ。

ro：double 类型，正态分布的方差 $\sigma^2=ro^2$ 值。

seed:double 类型引用,存放随机数种子初值,再返回一个新值。

dp:double 类型引用,存放生成随机数序列。

stCount:类型为非负整数,指定要生成的随机数序列长度。

5. 返回值

无。

6. 函数说明

rand_NormalDistributing_One()函数的参数为 mu 和 ro(μ 和 σ 值),要事先给定,也可从其他程序中获得。

如果算法中的 n 值要取得更大(超过 12),rand_NormalDistributing_One()程序可根据正态分布随机数 y 的计算公式稍加修改。

7. 示例

分别取 μ 值和 σ 值为 1.0 和 1.5,生成 40 个正态分布的随机数。

主函数及示例程序如下:

```
//rand_NormalDistributing_Series.cpp  产生任意均值与方差的正态分布随机数序列

#include <random.h>   //随机算法头文件
#include <iostream>   //输入输出流头文件
#include <time.h>     //时间头文件
using namespace std;  //名字空间

//产生 40 个均值与方差的正态分布随机数
void main(void)
{
  double mu = 1.0, ro = 1.5;
  valarray<double> rip(40);
  size_t stCount = 40;
  double seed = (double)time(NULL);  //取种子
  cout.setf(ios::fixed|ios::left);  //输出数据为定点法,靠左对齐
  cout.precision(6);                //精度 6 位
  //生成随机数
  rand_NormalDistributing_Series(mu, ro, seed, rip, stCount);
  ValarrayPrint(rip);               //输出至显示器
  cout << endl;
}
```

8. 示例结果

```
  2.239487    5.078720    0.494125    0.579453    0.928452
 -1.365128    1.292465    1.494980    2.336166    0.909775
  0.309555   -0.870743    1.962631    1.403427   -0.954605
  0.982285    1.307846    1.615829    0.499985    4.054062
  3.371811    0.046982   -1.326675    2.344589    0.654526
  2.696884    1.065414    1.853867   -0.844009   -0.934464
 -1.323746    1.081894    1.876205    2.652939   -0.994156
 -1.471329    2.815170    1.459091    0.554184    1.694199
```

第 4 章　多项式与连分式计算

本章介绍计算多项式与连分式值的算法。多项式的系数和自变量既可以是实数，也可以是复数，每种算法都用C++语言以函数的形式实现。4.1 节介绍多项式相关的基本概念，以及连分式和包含相关算法程序原型的头文件，4.2 节给出了计算各多项式与连分式函数值的使用示例及相关说明。

4.1　多项式与头文件

本节介绍多项式及相关基本概念，并包括多项式的加、减、乘、除等运算及连分式计算算法。其算法的函数程序原型包含在头文件 polynomials. h 中，函数的定义内容在 polynomials. inl 文件中。

本章介绍的多项式与连分式算法及程序包括：

- 计算一维多项式值；
- 计算一维多项式组值；
- 计算二维多项式值；
- 计算一维多项式相乘；
- 计算一维多项式相除；
- 计算连分式值。

4.1.1　基本概念

1. 数域

如果一个数系满足下列两个条件，则称这个数系为一个数域。

(1) 系中有不等于零的数；

(2) 对系内任意两个数(这两个数也可相同)的和、差、积、商(零不能作除数)仍为系内的数，这就是说，系内的数对于四则运算是封闭的。如有理数系、实数系、复数系都是数系。

2. 多项式

称

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x^1 + a_0 \qquad n > 0$$

为一元 n 次多项式，式中 n 为正整数，$a_n, a_{n-1}, \cdots, a_1, a_0$ 是属于数域 S 的常数，称为多项式的系数，最高次项系数 a_n 简称为首项系数。

3. 代数方程

形如

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x^1 + a_0 = 0 \qquad n > 0$$

的方程称为在数域 S 上的一个未知数的 n 次代数方程，其中 $a_n, a_{n-1}, \cdots, a_1, a_0$ 是属于数域 S 的常数。

4.1.2 polynomials.h

```
//polynomials.h        多项式与连分式头文件

#ifndef _POLYNOMIALS_H
#define _POLYNOMIALS_H

#include <comm.h>      //公共部分头文件
#include <math.h>      //数学头文件
#include <matrix.h>    //矩阵类头文件

//求一维实(复)多项式值
template<class T, class U>
inline U
PolyValueOneDim(valarray<T>& dCoff, size_t stNo, U dX);

//求一维多项式组值
template<class T, class V, class U>
inline void
PolyValueOneDimGroup(valarray<T>& dCoff, valarray<V>& dX, valarray<U>& dValue);

//求二维实(复)多项式值
template<class T, class U>
inline U //内联函数
PolyValueTwoDim(matrix<T>& dCoff, U dX, U dY);

//两个一维多项式相乘
template<class T>
inline void
PolyMultip(valarray<T>& dCoffP, valarray<T>& dCoffQ, valarray<T>& dCoffS);

//两个一维多项式除法
template<class T>
inline int
PolyDiv(valarray<T>& dCoffP, valarray<T>& dCoffQ, valarray<T>& dCoffS, valarray<T>&
        dCoffR);

// 计算连分式函数值
template<class T>
inline T
FractionValue(valarray<T>& dXpara, valarray<T>& dCoff, T dX);

#include "polynomials.inl" //多项式及连分式相关函数定义头文件
```

```
#endif //_POLYNOMIALS_H
```

4.2 多项式与连分式算法及程序

4.2.1 计算一维多项式值

1. 功能

计算一维实(复)多项式 $p(x)$在 x 处的值。

2. 算法

设 $n-1$ 次多项式为

$$p(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x^1 + a_0 \qquad n > 0$$

其中 n 为正整数,a_n与 x_n为实数或复数。

多项式求值采用我国宋代数学家秦九韶提出的算法,可以使用最少次数的乘法运算。国外称此算法为 Hornet 算法,但 Hornet 的工作比秦九韶晚了五六个世纪。多项式表述为如下形式:

$$p(x) = (\cdots((a_{n-1}x + a_{n-2})x + a_{n-3})x + \cdots + a_1)x + a_0$$

然后从里往外一层一层地进行计算。其递推公式如下:

$$u_{n-1} = a_{n-1} \qquad n > 0$$
$$u_k = u_{k+1}x + a_k \qquad k = n-2, \cdots, 1, 0$$

最后得到的 u_0 即是多项式值 $p(x)$。

3. 函数原型

```
//求一维实(复)多项式值
template<class T, class U>
inline U
PolyValueOneDim(valarray<T>& dCoff, size_t stNo, U dX);
```

4. 参数

dCoff:类型为 valarray<T>的引用,存放多项式系数。

stNo:非负整型。多项式项数,其最高次数为 stNo-1。

dX:类型为 U 的一维多项式自变量 x。

5. 返回值

类型为 U 的多项式值。

6. 函数说明

PolyValueOneDim()可求解以下几类一维多项式的值:多项式的系数和自变量值为实数;多项式的系数和自变量值为复数;多项式的系数为实数,自变量值为复数。具体应用见本节示例。

7. 示例

(1) 求一维实多项式

$$p(x) = 2.0x^6 - 5.0x^5 + 3.0x^4 + 1.0x^3 - 7.0x^2 + 7.0x - 20.0$$

当 x 在-0.9、0.9、-1.1、1.1、-1.3、1.3 点处的函数值。

(2) 求一维复多项式

$$p(z)=(1.2,3.4)z^6-(-3.2,-4.8)z^5+(2.7,-6.4)z^4+(3.1,5.9)z^3$$
$$-(3.5,3.9)z^2+(12.1,-5.4)z-(-2.5,8.4)$$

当点 z 为(−0.9,1.4)、(−0.5,1.4)、(0.9,−1.8)、(−1.1,−1.4)、(1.1,1.9)、(−1.5,1.9)处的函数值。

(3) 求系数为实数,自变量为复数的一维多项式:

$$p(z)=1.0z^6+2.0z^5+3.0z^4+4.0z^3+5.0z^2+6.0z+7.0$$

当点 z 为(−0.9,1.4)、(−0.5,1.4)、(0.9,−1.8)、(−1.1,−1.4)、(1.1,1.9)、(−1.5,1.9)处的函数值。

示例程序如下:

```
//PolyValueOneDim.cpp        一维实(复)多项式求值示例

#include <iostream>          //输入输出流头文件
#include <polynomials.h>     //多项式及连分式头文件
using namespace std;         //名字空间

void main(void)
{
  cout << "PolyValueOneDim()" << endl << endl;

  //多项式系数与自变量都为实数
  cout << "Coefficient and variable are real. Result is real: " << endl;
  double dCoff[7] = {-20.0, 7.0, -7.0, 1.0, 3.0, -5.0, 2.0};
  double x[6] = {0.9, -0.9, 1.1, -1.1, 1.3, -1.3};
  valarray<double> vCoff(dCoff,7);
  cout.setf(ios::fixed|ios::left);     //输出数据为定点法,靠左对齐
  cout.precision(6);                   //精度 6 位
  for(size_t st = 0; st < 6; st++)     //输出多项式值
  {
      double dValue = PolyValueOneDim(vCoff, 7, x[st]);
      cout << "x(" << st << ") = " << x[st];
      cout << "\t p(" << st << ") = " << dValue << endl;
  }
  cout << endl;

  //多项式系数与自变量都为复数
  cout << "Coefficient and variable are complex. Result is complex: " << endl;
  complex<double> cdCoff[7] = { complex<double>(1.2, 3.4),
          complex<double>(-3.2, -4.8),complex<double>(2.7, -6.4),
          complex<double>(3.1, 5.9), complex<double>(3.5, 3.9),
          complex<double>(12.1, -5.4), complex<double>(-2.5, 8.4),
  };
  complex<double> cx[6] = {
          complex<double>(0.9, 1.4), complex<double>(-0.5, 1.4),
```

```
            complex<double>(-0.9, -1.8),complex<double>(-1.1, -1.4),
            complex<double>(1.1, 1.9), complex<double>(-1.5, 1.9),
    };
    valarray<complex<double> > vcdCoff(cdCoff,7);
    for(st = 0; st < 6; st++) //输出多项式值
    {
        complex<double> cp = PolyValueOneDim(vcdCoff, 7, cx[st]);
        cout << "cx(" << st << ") = " << cx[st];
        cout << "\t cp(" << st << ") = " << cp << endl;
    }
    cout << endl;

    //多项式系数为实数,自变量为复数
    cout << "Coefficient is real. Variable is complex. Result is complex: " << endl;
    double drCoff[7] = {1,2,3,4,5,6,7};
    valarray<double> vdrCoff(drCoff,7);

    for(st = 0; st < 6; st++) //输出多项式值
    {
        complex<double> dp = PolyValueOneDim(vdrCoff, 7, cx[st]);
        cout << "cx(" << st << ") = " << cx[st];
        cout << "\t dp(" << st << ") = " << dp << endl;
    }
    cout << endl;
}
```

8. 示例结果

```
PolyValueOneDim()

Coefficient and variable are real. Result is real:
x(0) = 0.900000    p(0) = -18.562268
x(1) = -0.900000   p(1) = -26.715368
x(2) = 1.100000    p(2) = -19.556128
x(3) = -1.100000   p(3) = -21.513028
x(4) = 1.300000    p(4) = -20.875732
x(5) = -1.300000   p(5) = -6.340432

Coefficient and variable are complex. Result is complex:
cx(0) = (0.900000,1.400000)      cp(0) = (-18.235610,-37.414430)
cx(1) = (-0.500000,1.400000)     cp(1) = (-25.739103,132.825492)
cx(2) = (-0.900000,-1.800000)    cp(2) = (-452.432417,882.671463)
cx(3) = (-1.100000,-1.400000)    cp(3) = (317.078871,428.746330)
cx(4) = (1.100000,1.900000)      cp(4) = (-175.857919,110.482472)
cx(5) = (-1.500000,1.900000)     cp(5) = (-2043.362020,-339.453206)
```

```
Coefficient is real. Variable is complex. Result is complex:
cx(0) = (0.900000,1.400000)      dp(0) = (120.210275,-131.555116)
cx(1) = (-0.500000,1.400000)     dp(1) = (3.221123,-57.262100)
cx(2) = (-0.900000,-1.800000)    dp(2) = (291.012139,231.535548)
cx(3) = (-1.100000,-1.400000)    dp(3) = (139.025915,-88.703524)
cx(4) = (1.100000,1.900000)      dp(4) = (830.378720,-355.401384)
cx(5) = (-1.500000,1.900000)     dp(5) = (901.522208,674.201320)
```

9. 示例说明

在本节示例中，对求一维多项式的值，无论其系数与自变量是复数还是实数，其算法及公式都是一致的，这借助于C++中已经定义了复数模板类。有关复数的各种操作运算请参阅第2章“复数运算”。

4.2.2 计算一维多项式组值

1. 功能

利用系数预处理法对多项式

$$p(x)=a_{n-1}x^{n-1}+a_{n-2}x^{n-2}+\cdots+a_1x+a_0$$

进行多组求值。其中 $n=2^k(k\geqslant 1)$。

2. 算法

首先将多项式变为首一多项式(首项系数为1的多项式)，即令

$$T(x)=p(x)/a_{n-1}=x^{n-1}+t_{n-2}x^{n-2}+\cdots+t_1x+t_0$$

其中 $t^k=a_k/a_{n-1}(k=n-2,\cdots,1,0)$。

然后对首一多项式 $T(x)$的系数进行预处理。其预处理过程如下。

将 $T(x)$分解为如下形式：

$$T(x)=(x^j+b)q(x)+r(x)$$

其中 $j=2^{k-1}$，$q(x)$与 $r(x)$均为 $2^{k-1}-1$ 次的首一多项式。b、$q(x)$与 $r(x)$按以下规则确定。

多项式中的中项系数减1即为 b，即

$$b=t_{2^{k-1}}-1$$

多项式左半部分除以 x^j 即为 $q(x)$，即

$$q(x)=x^{s-1}+q_{s-2}x^{s-2}+\cdots+q_1x+q$$

其中 $s=2^{k-1}$，$q_i=t_{i+s}(i=s-2,\cdots,1,0)$。

多项式右半部分减去 b 与 $q(x)$的乘积即为 $r(x)$，即

$$r(x)=x^{s-1}+r_{s-2}x^{s-2}+\cdots+r_1x+r_0$$

式中 $r_i=t_i-bq_i(i=s-2,\cdots,1,0)$。

由于 $q(x)$与 $r(x)$还是首一多项式，因此，它们也可以按照上述方法分别进行分解。这个过程一直作到 $q(x)$与 $r(x)$的次数为1为止。在分解过程中，每次分解后的系数仍放在 $T(x)$的各系数的存储单元中，最后就得到 $T(x)$经分解处理后的系数。

首一多项式 $T(x)$的系数经预处理后，就可以用这些预处理后的系数对不同的 x 求函数值。这种方法特别适用于对多个 x 进行求值，减少求值中的乘法次数。

如果原来的多项式不满足 $n=2^k$ 这个条件，则可以添加系数为0的各项及系数为1的最高次项。

3. 函数原型

```
template<class T, class V, class U>
inline void
PolyValueOneDimGroup(valarray<T>& dCoff,
        valarray<V>& dX, valarray<U>& dValue);
```

4. 参数

dCoff：类型为 valarray<T>的引用，长度为 n，存放 $n-1$ 次多项式的系数 $a_{n-1}, a_{n-2}, \cdots, a_1, a_0$。返回时该数组将被破坏。

dX：类型为 valarray<V>的引用，长度为 m。存放 m 个自变量值。

dValue：类型为 valarray<U>的引用，长度为 m，返回与给定 n 个自变量值对应的多项式值，即

$$p_k = p(x_k), k = 0, 1, \cdots, m-1$$

5. 返回值

无。

6. 函数说明

PolyValueOneDimGroup()可一次求解多个多项式值，具体应用见本节示例。

7. 示例

(1) 利用系数预处理法计算多项式

$$p(x) = 2x^6 + 5x^5 + 3x^4 + x^3 - 7x^2 + 7x - 20$$

在 $x=\pm 0.9, \pm 1.1, \pm 1.3$ 处的函数值。

(2) 利用系数预处理法计算多项式

$$p(x) = \sum_{k=1}^{16} kx^{k-1}$$

在 $x=\pm 0.9, \pm 1.1, \pm 1.3$ 处的函数值。

示例程序如下：

```
//PolyValueOneDimGroup.cpp 求一维多项式组值示例

#include <iostream>        //输入输出流头文件
#include <polynomials.h>   //多项式及连分式头文件
using namespace std;       //名字空间

//求多项式 p(x,y)在多点处的函数值
void main(void)
{
  cout << "PolyValueOneDimGroup()" << endl << endl;
  cout << "\t Group 1 " << endl; //第一组求值
  double da[7] = {-20.0, 7.0, -7.0, 1.0, 3.0, -5.0, 2.0};
  double dx[6] = {0.9, -0.9, 1.1, -1.1, 1.3, -1.3};
  valarray<double> vda(da,7);
  valarray<double> vdx(dx,6);
  valarray<double> vdp(6);
  PolyValueOneDimGroup(vda, vdx, vdp);
```

```
    cout << endl;
    for(size_t st = 0; st<6; st + + )
    {
        cout << "x(" << st << ") = " << vdx[st];
        cout << "\t p(" << st << ") = "<< vdp[st] << endl;
    }
    cout << endl << "\t Group 2 " << endl; //第二组求值
    double db[16] =
    {
       1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
       9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0
    };
    valarray<double> vdb(db,16);
    PolyValueOneDimGroup(vdb, vdx, vdp);
    cout << endl;
    for(st = 0; st<6; st + + )
    {
        cout << "x(" << st << ") = " << vdx[st];
        cout << "\t p(" << st << ") = " << vdp[st] << endl;
    }
    cout << endl;
}
```

8. 示例结果

```
PolyValueOneDimGroup()

  Group 1

x(0) = 0.900000    p(0) = -18.562268
x(1) = -0.900000   p(1) = -26.715368
x(2) = 1.100000    p(2) = -19.556128
x(3) = -1.100000   p(3) = -21.513028
x(4) = 1.300000    p(4) = -20.875732
x(5) = -1.300000   p(5) = -6.340432

  Group 2

x(0) = 0.900000    p(0) = 51.821475
x(1) = -0.900000   p(1) = -1.334760
x(2) = 1.100000    p(2) = 375.698379
x(3) = -1.100000   p(3) = -35.824505
x(4) = 1.300000    p(4) = 2820.647905
x(5) = -1.300000   p(5) = -475.288239
```

4.2.3 计算二维多项式值

1. 功能

计算二维多项式 $p(x,y)$在(x,y)处的值。

2. 算法

设二维多项式为如下形式：

$$p(x,y)=\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}a_{ij}x^iy^j=\sum_{i=0}^{m-1}\left[\sum_{j=0}^{n-1}(a_{ij}x^i)y^j\right]\qquad m>0,n>0$$

令

$$S_i=\sum_{j=0}^{n-1}(a_{ij}x^i)y^j\qquad i=0,1,\cdots,m-1$$

则递推公式如下：

$$u_{n-1}=a_{i,n-1}x^i\qquad n>0$$

$$u_j=u_{j+1}y+a_{ij}x^i\qquad j=n-2,\cdots,1,0$$

其中最后得到的 u_0 即为 S_i。

最后将所有的 $S_i(i=0,1,\cdots,m-1)$累加，即有：

$$p(x,y)=\sum_{i=0}^{m-1}S_i$$

得到的结果就是二维多项式在(x,y)处的函数值。

算法中 i,j,n 和 m 为正整数，a_{ij} 与 x 和 y 为实数或复数。

3. 函数原型

```
//求二维实(复)多项式值
template<class T, class U>
inline U  //内联函数
PolyValueTwoDim(matrix<T>& dCoff, U dX, U dY);
```

4. 参数

dCoff：类型为 matrix<T>引用，存放二维多项式系数。

dX：类型为 U 的二维多项式自变量 x。

dY：类型为 U 的二维多项式自变量 y。

5. 返回值

类型为 U 的二维多项式值。

6. 函数说明

PolyValueTwoDim()可求解以下几类二维多项式的值：多项式的系数和自变量值为实数；多项式的系数和自变量值为复数；多项式的系数为实数，自变量值为复数。具体应用见本节示例。

7. 示例

(1) 计算二维多项式

$$p(x,y)=\sum_{i=0}^{3}\sum_{j=0}^{4}a_{ij}x^iy^j$$

在点(dx,dy)为(0.6081，−1.3054)，(0.7182，−1.4155)，(0.8283，−1.5256)，(0.9384，−1.6357)，(1.0485，−1.7458)，(1.1586，−1.8559)处的函数值。

其中，系数矩阵为：

$$\boldsymbol{A}=\begin{bmatrix}1.0 & 2.0 & 3.0 & 4.0 & 5.0\\ 6.0 & 7.0 & 8.0 & 9.0 & 10.0\\ 11.0 & 12.0 & 13.0 & 14.0 & 15.0\\ 16.0 & 17.0 & 18.0 & 19.0 & 20.0\end{bmatrix}$$

(2) 计算多项式

$$p(x,y)=\sum_{i=0}^{4}\sum_{j=0}^{2}a_{ij}x^iy^j$$

在点(cdx,cdy)为((0.11,−1.99)，(−0.11,1.99))，((0.22,−1.88)，(−0.22,1.88))，((0.33,−1.77)，(−0.33,1.77))，((0.44,−1.66)，(−0.44,1.66))，((0.55,−1.55)，(−0.55,1.55))，((0.66,−1.44)，(−0.66,0.59))处的函数值。

其中，系数矩阵为：

$$\boldsymbol{A}=\begin{bmatrix}(1.2,3.4) & (-3.2,-4.8) & (2.7,-6.4)\\ (3.1,-0.9) & (-3.5,1.9) & (2.1,-5.4)\\ (-2.5,1.8) & (-2.4,2.4) & (-1.5,0.4)\\ (-2.5,2.1) & (-2.5,8.4) & (2.3,-1.4)\\ (1.1,-0.2) & (-0.4,3.4) & (0.5,-2.4)\end{bmatrix}$$

(3) 计算二维多项式

$$p(x,y)=\sum_{i=0}^{4}\sum_{j=0}^{2}a_{ij}x^iy^j$$

在点(cdx,cdy)为((0.11,−1.99)，(−0.11,1.99))，((0.22,−1.88)，(−0.22,1.88))，((0.33,−1.77)，(−0.33,1.77))，((0.44,−1.66)，(−0.44,1.66))，((0.55,−1.55)，(−0.55,1.55))，((0.66,−1.44)，(−0.66,0.59))处的函数值。

其中，系数矩阵为：

$$\boldsymbol{A}=\begin{bmatrix}1.0 & 2.0 & 3.0\\ 4.0 & 5.0 & 6.0\\ 7.0 & 8.0 & 9.0\\ 10.0 & 11.0 & 12.0\\ 13.0 & 14.0 & 15.0\end{bmatrix}$$

示例程序如下：

```
//PolyValueTwoDim.cpp 二维实(复)多项式求值示例

#include <iostream>          //输入输出流头文件
#include <polynomials.h>     //多项式及连分式头文件
using namespace std;         //名字空间

//求多项式 p(x,y)在多点处的函数值
void main(void)
{
  cout << "PolyValueTwoDim()" << endl << endl;
```

```
//多项式系数与自变量都为实数
cout << "Coefficient and variable are real. Result is real: " << endl;
double dCoff[4][5] =
{
    { 1.0, 2.0, 3.0, 4.0, 5.0},
    { 6.0, 7.0, 8.0, 9.0, 10.0},
    {11.0, 12.0, 13.0, 14.0, 15.0},
    {16.0, 17.0, 18.0, 19.0, 20.0}
};

matrix<double> mdCoff(&dCoff[0][0],4,5);
double dX = 0.6081, dY = -1.3054;
cout.setf(ios::fixed|ios::left);     //输出数据为定点法,靠左对齐
cout.precision(4);                   //精度 4 位
for(size_t st = 0; st < 6; st++)  //输出多项式值
{
    double dValue = PolyValueTwoDim(mdCoff, dX, dY);
    cout << "p(" << dX << ", " << dY << ") = " << dValue << endl;
    dX = dX + 0.1101;
    dY = dY - 0.1101;
}
cout << endl;

//多项式系数与自变量都为复数
cout << "Coefficient and variable are complex. Result is complex: " << endl;
complex<double> cdCoff[5][3] =
{
    {complex<double>(1.2, 3.4),complex<double>(-3.2, -4.8),
     complex<double>(2.7, -6.4)},
    {complex<double>(3.1, -0.9),complex<double>(-3.5, 1.9),
     complex<double>(2.1, -5.4)},
    {complex<double>(-2.5, 1.8),complex<double>(-2.4, 2.4),
     complex<double>(-1.5, 0.4)},
    {complex<double>(-2.5, 2.1),complex<double>(-2.5, 8.4),
     complex<double>(2.3, -1.4)},
    {complex<double>(1.1, -0.2),complex<double>(-0.4, 3.4),
     complex<double>(0.5, -2.4)},
};
matrix<complex<double> > mcdCoff(&cdCoff[0][0],5,3);
complex<double> cdX = complex<double>( 0.11, -1.99);
complex<double> cdY = complex<double>(-0.11, 1.99);
for(st = 0; st < 6; st++) //输出多项式值
{
    complex<double> cp = PolyValueTwoDim(mcdCoff, cdX, cdY);
```

```
        cout << "cp(cdX, cdY) = cp(" << cdX << ", " << cdY << ") = " << cp << endl;
        cdX = cdX + complex<double>(0.11, 0.11);
        cdY = cdY - complex<double>(0.11, 0.11);
    }
    cout << endl;
    //多项式系数为实数,自变量为复数
    cout << "Coefficient is real. Variable is complex. Result is complex: " << endl;
    double dcCoff[5][3] =
    {
        { 1.0, 2.0, 3.0},
        { 4.0, 5.0, 6.0},
        { 7.0, 8.0, 9.0},
        {10.0, 11.0, 12.0},
        {13.0, 14.0, 15.0}
    };
    matrix<double> mdcCoff(&dcCoff[0][0], 5,3);
    cdX = complex<double>( 0.11, -1.99);
    cdY = complex<double>(-0.11, 1.99);
    for(st = 0; st < 6; st++) //输出多项式值
    {
        complex<double> dp = PolyValueTwoDim(mdcCoff, cdX, cdY);
        cout << "dp(cdX, cdY) = dp(" << cdX << ", " << cdY << ") = " << dp << endl;
        cdX = cdX + complex<double>(0.11, 0.11);
        cdY = cdY - complex<double>(0.11, 0.11);
    }
}
```

8. 示例结果

```
PolyValueTwoDim()

Coefficient and variable are real. Result is real:
p(0.6081, -1.3054) = 41.1067
p(0.7182, -1.4155) = 73.1972
p(0.8283, -1.5256) = 126.5385
p(0.9384, -1.6357) = 212.4934
p(1.0485, -1.7458) = 347.0377
p(1.1586, -1.8559) = 552.0781

Coefficient and variable are complex. Result is complex:
cp(cdX, cdY) = cp((0.1100,-1.9900), (-0.1100,1.9900)) = (-105.8723,-71.3821)
cp(cdX, cdY) = cp((0.2200,-1.8800), (-0.2200,1.8800)) = (-55.2398,-93.1733)
cp(cdX, cdY) = cp((0.3300,-1.7700), (-0.3300,1.7700)) = (-7.0819,-94.4875)
cp(cdX, cdY) = cp((0.4400,-1.6600), (-0.4400,1.6600)) = (33.1850,-79.0203)
cp(cdX, cdY) = cp((0.5500,-1.5500), (-0.5500,1.5500)) = (61.5554,-51.3128)
cp(cdX, cdY) = cp((0.6600,-1.4400), (-0.6600,1.4400)) = (75.4757,-16.4707)
```

```
Coefficient is real. Variable is complex. Result is complex:
dp(cdX, cdY) = dp((0.1100, -1.9900), (-0.1100,1.9900)) = (-765.7566, -150.9798)
dp(cdX, cdY) = dp((0.2200, -1.8800), (-0.2200,1.8800)) = (-463.2483, -296.7640)
dp(cdX, cdY) = dp((0.3300, -1.7700), (-0.3300,1.7700)) = (-211.6429, -325.8572)
dp(cdX, cdY) = dp((0.4400, -1.6600), (-0.4400,1.6600)) = (-25.5946, -276.0078)
dp(cdX, cdY) = dp((0.5500, -1.5500), (-0.5500,1.5500)) = (89.6972, -181.5440)
dp(cdX, cdY) = dp((0.6600, -1.4400), (-0.6600,1.4400)) = (137.0428, -72.4331)
```

9. 示例说明

在本节示例中求二维多项式的值，无论其系数与自变量是复数还是实数，其算法及公式都是一致的，这借助于C++中已经定义了复数模板类。有关复数的各种操作运算请参阅第2章“复数运算”。

4.2.4 计算一维多项式相乘

1. 功能

求两一维实(复)多项式相乘的积。

2. 算法

设两个一维多项式为：

$$P(x) = p_{m-1}x^{m-1} + p_{m-2}x^{m-2} + \cdots + p_1x^1 + p_0 \qquad m > 0$$

$$Q(x) = q_{n-1}x^{n-1} + q_{n-2}x^{n-2} + \cdots + q_1x^1 + q_0 \qquad n > 0$$

其乘积多项式为：

$$\begin{aligned} S(x) &= P(x)Q(x) \\ &= s_{m+n-2}x^{m+n-2} + \cdots + s_2x^2 + s_1x^1 + s_0 \end{aligned}$$

其中，m、n 为正整数，p_m、q_n、x 为实数。

乘积多项式 $S(x)$中的各系数按如下公式计算：

$$s_k = 0, k = 0,1,\cdots,m+n-2$$

$$s_{i+j} = s_{i+j} + p_iq_j, \qquad i = 0,1,\cdots,m-1; j = 0,1,\cdots,n-1$$

3. 函数原型

```
//两个一维多项式相乘
template<class T>
inline void
PolyMultip(valarray<T>& dCoffP, valarray<T>& dCoffQ, valarray<T>& dCoffS);
```

4. 参数

dCoffP：类型为 valarray<T>的引用，存放被乘多项式 $P(x)$系数。

dCoffQ：类型为 valarray<T>的引用，存放乘多项式 $Q(x)$系数。

dCoffS：类型为 valarray<T>的引用，存放积多项式 $S(x)$系数。

5. 返回值

无。

6. 函数说明

PolyMultip()计算两多项式的乘积多项式。被乘多项式与乘多项式的系数可以是实数，也可以是复数。最后结果得到积多项式的系数，积多项式系数的类型依被乘多项式与乘多项

式的系数而定。具体应用见本节示例。

7. 示例

(1) 求两多项式

$$P(x) = 4.2x^5 - 6.1x^4 + 5.4x^3 + 2.5x^2 - 1.8x + 3.6$$

与

$$Q(x) = 2.92x^3 + 3.71x^2 - 6.45x + 2.64$$

的乘积多项式 $S(x)$。

(2) 计算两复多项式

$$P(x) = (1.2,3.4)x^6 + (-3.2,-4.8)x^5 + (2.7,-6.4)x^4 + (3.1,5.9)x^3 + (3.5,3.9)x^2 + (12.1,-5.4)x + (-2.5,8.4)$$

与多项式

$$Q(x) = (0.9,1.4)x^5 + (-0.5,1.4)x^4 + (-0.9,-1.8)x^3 + (-1.1,-1.4)x^2 + (1.1,1.9)x + (-1.5,1.9)$$

的积多项式 $S(x)$。

示例程序如下：

```
//PolyMultip.cpp     两个一维实(复)多项式相乘示例

#include <iostream>        //输入输出流头文件
#include <polynomials.h>   //多项式头文件
using namespace std;       //名字空间

void main()
{
  cout << "PolyMultip()" << endl << endl;

  //多项式系数为实数
  cout << "Coefficients are real. Result is real: " << endl;
  double dCoffP[6] = {4.2, -6.1, 5.4, 2.5, -1.8, 3.6};
  double dCoffQ[4] = {2.92, 3.71, -6.45, 2.64};
  valarray<double> vdCoffP(dCoffP,6);
  valarray<double> vdCoffQ(dCoffQ,4);
  valarray<double> vdCoffS(9);
  cout.setf(ios::fixed|ios::left);  //输出数据为定点法,靠左对齐
  cout.precision(4);                //精度 4 位
  PolyMultip(vdCoffP, vdCoffQ, vdCoffS);
  for (size_t st = 0; st < 9; st++)
  {
      cout << "dCoffS(" << st << ") = " << vdCoffS[st] << endl;
  }
  cout << endl;

  //多项式系数为复数
```

```
    cout << "Coefficients are complex. Result is complex: " << endl;
    complex<double> cdCoffP[7] =
    {
        complex<double>(1.2, 3.4),
        complex<double>(-3.2, -4.8), complex<double>(2.7, -6.4),
        complex<double>(3.1, 5.9), complex<double>(3.5, 3.9),
        complex<double>(12.1, -5.4), complex<double>(-2.5, 8.4),
    };
    complex<double> cdCoffQ[6] =
    {
        complex<double>(0.9, 1.4), complex<double>(-0.5, 1.4),
        complex<double>(-0.9, -1.8), complex<double>(-1.1, -1.4),
        complex<double>(1.1, 1.9), complex<double>(-1.5, 1.9),
    };
    valarray<complex<double> > vcdCoffP(cdCoffP,7);
    valarray<complex<double> > vcdCoffQ(cdCoffQ,6);
    valarray<complex<double> > vcdCoffS(12);
    //complex<double> cdCoffS[12];
    PolyMultip(vcdCoffP, vcdCoffQ, vcdCoffS);
    for (st = 0; st < 12; st++)
    {
        cout << "cdCoffS(" << st << ") = " << vcdCoffS[st] << endl;
    }
    cout << endl;
}
```

8. 示例结果

```
PolyMultip()

Coefficients are real. Result is real:
dCoffS(0) = 12.2640
dCoffS(1) = -2.2300
dCoffS(2) = -33.9530
dCoffS(3) = 77.7670
dCoffS(4) = -46.9150
dCoffS(5) = 1.9650
dCoffS(6) = 31.5660
dCoffS(7) = -27.9720
dCoffS(8) = 9.5040

Coefficients are complex. Result is complex:
cdCoffS(0) = (-3.6800,4.7400)
cdCoffS(1) = (-1.5200,-8.8200)
cdCoffS(2) = (24.7500,-9.2800)
cdCoffS(3) = (-0.1800,21.2900)
```

```
cdCoffS(4) = (-34.4100,26.4800)
cdCoffS(5) = (4.4800,-6.7800)
cdCoffS(6) = (25.2700,2.2700)
cdCoffS(7) = (-29.2000,-6.7000)
cdCoffS(8) = (-22.9200,-6.0800)
cdCoffS(9) = (25.4200,12.1100)
cdCoffS(10) = (-26.6000,35.5800)
cdCoffS(11) = (-12.2100,-17.3500)
```

9. 示例说明

在本节示例中对求两一维多项式相乘的值，无论它们的系数是复数还是实数，其算法及公式都是一致的，这借助于C++中已经定义了的复数模板类。有关复数的各种操作运算请参阅第2章“复数运算”。

4.2.5 计算一维多项式相除

1. 功能

求两个一维多项式除法的商多项式和余多项式。

2. 算法

设多项式

$$P(x) = p_{m-1}x^{m-1} + p_{m-2}x^{m-2} + \cdots + p_1x^1 + p_0 \qquad m > 0$$

被多项式

$$Q(x) = q_{n-1}x^{n-1} + q_{n-2}x^{n-2} + \cdots + q_1x^1 + q_0 \qquad n > 0$$

除得的商多项式与余多项式分别为 $S(x)$和 $R(x)$。采用综合除法求商多项式 $S(x)$中的各系数。

设商多项式 $S(x)$的最高次数为 $k=m-n$，则 $S(x)$的系数由如下过程来确定($i=0,1,\cdots,k$)：

$$s_{k-i} = p_{m-1-i}/q_{n-1}$$

$$p_j = p_j - s_{k-i}q_{j+i-k}, j = m-i-1,\cdots,k-i$$

最后的 $p_0, p_1, \cdots, p_{n-2}$即为余多项式的系数 $r_0, r_1, \cdots, r_{n-2}$。

3. 函数原型

```
//两个一维多项式除法
template<class T>
inline int
PolyDiv(valarray<T>& dCoffP, valarray<T>& dCoffQ, valarray<T>& dCoffS,
                                                  valarray<T>& dCoffR);
```

4. 参数

dCoffP：类型为 valarray<T>的引用，存放被除多项式 $P(x)$系数。

dCoffQ：类型为 valarray<T>的引用，存放除多项式 $Q(x)$系数。

dCoffS：类型为 valarray<T>的引用，存放商多项式 $S(x)$系数。

dCoffR：类型为 valarray<T>的引用，存放余多项式 $R(x)$系数。

5. 返回值

int 类型。

0：除多项式 $Q(x)$的最高项系数为0，程序失败，退出；

1:程序成功,正常返回。

6. 函数说明

PolyDiv()计算两多项式相除后产生的商多项式和余多项式。被除多项式与除多项式的系数可以是实数,也可以是复数。最后结果得到商多项式和余多项式的系数。商多项式和余多项式系数的类型依被除多项式与除多项式的系数而定。具体应用见本节示例。

7. 示例

(1) 计算被除多项式

$$P(x) = 8.0x^4 - 5.0x^3 - 3.0x^2 + 6.0x + 3.0$$

与除多项式

$$Q(x) = 1.0x^2 - 1.0x + 2.0$$

的商多项式 $S(x)$和余多项式 $R(x)$。

(2) 计算复多项式

$$P(x) = (8.0,3.0)x^4 + (-5.0,4.0)x^3 + (-3.0,4.0)x^2 + (6.0,-5.0)x + (3.0,-1.0)$$

除以多项式

$$Q(x) = (1.0,2.0)x^2 + (-1.0,-3.0)x + (2.0,2.0)$$

的商多项式 $S(x)$与余多项式 $R(x)$。

示例程序如下:

```
//PolyDiv.cpp            一维多项式除法示例

#include <iostream>       //输入输出流头文件
#include <polynomials.h>  //多项式头文件
using namespace std;      //名字空间

void main()
{
  cout << "PolyDiv()" << endl << endl;

  //多项式系数为实数
  cout << "Coefficients are real. Result is real: " << endl;
  double dCoffP[5] = {8.0,-5.0,-3.0,6.0,3.0};
  double dCoffQ[3] = {1.0,-1.0,2.0};
  valarray<double> vdCoffP(dCoffP,5), vdCoffQ(dCoffQ,3);
  valarray<double> vdCoffS(3), vdCoffR(2);

  cout.setf(ios::fixed|ios::left);  //输出数据为定点法,靠左对齐
  cout.precision(4);                //精度 4 位
   if(PolyDiv(vdCoffP, vdCoffQ, vdCoffS, vdCoffR) > 0)
  {
      for(size_t st = 0; st < 3; st++)
          cout << "dCoffS(" << st << ") = " << vdCoffS[st] << endl;
      cout << endl;
      for(st = 0; st < 2; st++)
```

```
            cout << "dCoffR(" << st << ") = " << vdCoffR[st] << endl;
    }
    cout << endl;

    //多项式系数为复数
    cout << "Coefficients are complex. Result is complex: " << endl;
    complex<double> cdCoffP[5] =
    {
        complex<double>( 8.0, 3.0),
        complex<double>(-5.0, 4.0), complex<double>(-3.0, 4.0),
        complex<double>( 6.0, -5.0), complex<double>( 3.0, -1.0)
    };
    complex<double> cdCoffQ[3] =
    {
        complex<double>( 1.0, 2.0),
        complex<double>(-1.0, -3.0), complex<double>(2.0, 2.0)
    };
    valarray<complex<double> > vcdCoffP(cdCoffP,5), vcdCoffQ(cdCoffQ,3);
    valarray<complex<double> > vcdCoffS(3), vcdCoffR(2);
    if(PolyDiv(vcdCoffP, vcdCoffQ, vcdCoffS, vcdCoffR) > 0)
    {
       for (size_t st = 0; st < 3; st++)
           cout << "cdCoffS(" << st << ") = " << vcdCoffS[st] << endl;
       cout << endl;
       for (st = 0; st < 2; st++)
           cout << "cdCoffR(" << st << ") = " << vcdCoffR[st] << endl;
    }
}
```

8. 示例结果

```
PolyDiv()

Coefficients are real. Result is real:
dCoffS(0) = -0.3750
dCoffS(1) = 3.7500
dCoffS(2) = 1.5000

dCoffR(0) = 8.3750
dCoffR(1) = -9.1250

Coefficients are complex. Result is complex:
cdCoffS(0) = (2.6250,-0.5000)
cdCoffS(1) = (1.2500,-3.5000)
cdCoffS(2) = (0.5000,-1.0000)
```

```
cdCoffR(0) = (4.3750,-1.7500)
cdCoffR(1) = (-9.1250,12.3750)
```

9. 示例说明

在本节示例中，对求两一维多项式相除的值，无论它们的系数是复数还是实数，其算法及公式都是一致的，这借助于C++中已经定义了的复数模板类。有关复数的各种操作运算请参阅第2章“复数运算”。

4.2.6 计算连分式值

1. 功能

计算连分式函数

$$p(x)=b_0+\cfrac{x-x_0}{b_1+\cfrac{x-x_1}{b_2+\cdots+\cfrac{x-x_{n-2}}{b_{n-1}}}}$$

在指定处 x 的函数值。

2. 算法

从最后一节开始，逐节往前递推计算。即

$$\begin{cases}u=b_{n-1}\\ u=b_k+(x-x_k)/u & k=n-2,\cdots,1,0\end{cases}$$

最后的 u 就是连分式的值。其中 n 为正整数，b_k 与 x_k 为实数或复数。

3. 函数原型

```
// 计算连分式函数值
template<class T>
inline T
FractionValue(valarray<T>& dXpara, valarray<T>& dCoff, T dX);
```

4. 参数

dXpara：类型为 valarray<T>引用，存放连分式中各结点 $x_k(k=0,1,2,\cdots,n-1)$ 的值，其中 x_{n-1} 为任意。

dCoff：类型为 valarray<T>引用，存放连分式中各结点 $b_k(k=0,1,2,\cdots,n-1)$ 的值。

dX：类型为 T 的一维多项式自变量 x 的值。

5. 返回值

类型为 T 的连分式函数值。

6. 函数说明

函数输入输出值应为浮点型。

7. 示例

示例计算连分式函数

$$p(x)=1+\cfrac{x-1}{3+\cfrac{x-2}{-1+\cfrac{x-3}{2+\cfrac{x-4}{5+\cfrac{x-5}{-8+\cfrac{x-6}{11}}}}}}$$

在 $x=0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5$ 处的函数值。

示例程序如下：

```
//FractionValue.cpp                计算函数连分式值

#include <iostream>          //输入输出流头文件
#include <polynomials.h>     //多项式头函数
using namespace std;         //名字空间

void main()
{
  cout << "FractionValue()" << endl <<endl;
  double dXpara[7] = {1.0,2.0,3.0,4.0,5.0,6.0,0.0};
  double dCoff[7] = {1.0,3.0,-1.0,2.0,5.0,-8.0,11.0};
  valarray<double> vdXpara(dXpara,7), vdCoff(dCoff,6);
  double x = 0.0, dValue;
  cout.setf(ios::fixed|ios::left);  //输出数据为定点法,靠左对齐
  cout.precision(6);                //精度 6 位
  for(size_t st = 0; st<8; st++)
  {
      dValue = FractionValue(vdXpara,vdCoff,x);
      cout << "x = "<< x << "\t y = " << dValue << endl;
      x = x + 0.5;
  }
}
```

8. 示例结果

```
FractionValue()

x = 0.000000   y = 0.720000
x = 0.500000   y = 0.857678
x = 1.000000   y = 1.000000
x = 1.500000   y = 1.153846
x = 2.000000   y = 1.333333
x = 2.500000   y = 1.573913
x = 3.000000   y = 2.000000
x = 3.500000   y = 3.592593
```

第5章 矩阵运算

本章遵循C++的标准定义了矩阵模板类 class matrix。在该模板中定义和重载了许多与矩阵运算相关的操作符及运算，分别存放在 matrix.h 及 matrix.inl 文件中。编程人员可以直接利用已定义好的该模板类和相关的操作，很方便地对矩阵进行各种运算操作。

本章首先简要介绍了有关矩阵及矩阵运算的基本概念，然后分别介绍矩阵模板类 class matrix 及相关内容，随后通过示例介绍怎样利用这些内容定义矩阵类的对象，以及对矩阵进行各种基本的运算操作，最后介绍的是关于矩阵的其他一些常用运算操作。

5.1 矩阵概念

5.1.1 矩阵

1. 定义

数域 F 上的 $m\times n$ 个数 $a_{ij}(i=1,2,\cdots,m,j=1,2,\cdots,n)$ 按确定的位置排成矩形阵列，称为 $m\times n$ 矩阵，记作

$$\boldsymbol{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}_{m\times n}$$

其中横的一排叫做行，竖的一排叫做列，a_{ij} 称为矩阵的第 i 行第 j 列的元(素)。矩阵 $\boldsymbol{A}$ 简记为 (a_{ij}) 或 $(a_{ij})_{m\times n}$。数域 F 常为实数域或复数域，则称 $\boldsymbol{A}$ 为 $m\times n$ 实(矩)阵或复(矩)阵。当 $n=m$ 时，称 $\boldsymbol{A}$ 为 n 阶方阵。

2. 向量的线性相关与线性无关

对于 n 维空间的一组向量 $x_1,x_2,\cdots,x_m$，若数域 F 中有一组不全为零的数 $k_i(i=1,2,\cdots,m)$，使

$$k_1x_1+k_2x_2+\cdots+k_mx_m=0$$

成立，则称为组向量在 F 上线性相关，否则称这组向量在 F 上线性无关。

3. 行向量与列向量

由矩阵任一行的元素构成的 n 维向量称为行向量，记为

$$a_i=(a_{i1},a_{i2},\cdots,a_{in}) \qquad (i=1,2,\cdots,m)$$

由矩阵任一列的元素构成的 m 维向量称为列向量，记为

$$a_j=\begin{bmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{bmatrix}=(a_{1j},a_{2j},\cdots,a_{mj})^{\mathrm{T}} \qquad (j=1,2,\cdots,n)$$

式中 T 表示转置，即行(列)转换为列(行)。

4. 矩阵的秩

若矩阵 A 的 n 个列向量中有 r 个线性无关($r\leqslant n$)，而所有个数大于 r 的列向量组都线性相关，则称数 r 为矩阵 A 的列秩。类似可定义矩阵 A 的行秩。

矩阵 A 的列秩与行秩一定相等，它也称为矩阵 A 的秩，记作 $rankA=r$。

矩阵的秩也等于该矩阵中不等于零的子式的最大阶数。

5. 行列式

设

$$D=\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}$$

是由排成 n 阶方阵形式的 n^2 个数 $a_{ij}(i,j=1,2,\cdots,n)$ 确定的一个数，其值为 $n!$ 项之和

$$D=\sum(-1)^k a_{1k_1}a_{1k_2}\cdots a_{1k_n}$$

式中 $k_1,k_2,\cdots,k_n$ 是将序列 $1,2,\ldots,n$ 的元素次序交换 k 次所得到的一个序列，$\sum$ 号表示对 $k_1,k_2,\cdots,k_n$ 取遍 $1,2,\cdots,n$ 的一切排列求和，那么数 D 称为 n 阶方阵相应的行列式。

若 n 阶方阵 $A=(a_{ij})$，则 A 相应的行列式 D 记作

$$D=|A|=\det A=\det(a_{ij})$$

若矩阵 A 相应的行列式 $D=0$，称 A 为奇异矩阵，否则称为非奇异矩阵。

5.1.2 矩阵基本运算

1. 矩阵加矩阵

$C=A+B$，$c_{ij}=a_{ij}+b_{ij}$，$i=1,2\ldots,m$；$j=1,2,\ldots,n$

2. 矩阵减矩阵

$C=A-B$，$c_{ij}=a_{ij}-b_{ij}$，$i=1,2\ldots,m$；$j=1,2,\ldots,n$

3. 矩阵乘矩阵

$C_{m\times n}=A_{m\times r}\times B_{r\times n}$，$c_{ij}=\sum_{k=1}^{r}a_{ik}\cdot b_{kj}$，$m,r,n=1,2,\ldots$

4. 矩阵与数的运算

$C=d\&A$，$c_{ij}=d\&a_{ij}$，或 $c_{ij}=a_{ij}\&d$，$i=1,2\ldots,m$；$j=1,2,\ldots,n$

其中 d 为实(复)数，$\&$ 为加、减、乘或除算子。

5.1.3 几类特殊矩阵

1. 对角阵

设 A 为方阵，a_{ii} 称为对角元($i=1,2,\ldots,n$)，若对角元中至少一个不为 0，其余非对角元全为 0，则称 A 矩阵为“对角阵”。特别地，若对角阵中的对角元全为 1，称 A 矩阵为“单位阵”，常记作 I。

若对角阵中对角元的两侧也有非 0 元素，其余元素都为 0，则称其为“三对角阵”，如下面的 B 矩阵。

$$A=\begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix} \qquad B=\begin{bmatrix} b_{11} & b_{22} & 0 & \cdots & 0 \\ b_{21} & b_{22} & b_{23} & \cdots & 0 \\ 0 & b_{32} & b_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & b_{n-1,n} \\ 0 & 0 & 0 & \cdots & b_{nn} \end{bmatrix}$$

2. 上三角阵与下三角阵

设 A 为方阵，若其元素 $a_{ij}(i>j)$ 不全为 0，$a_{ij}(i<j)$ 全为 0，则构成的矩阵称为“下三角阵”，常记作 L；若其元素 $a_{ij}(i<j)$ 不全为 0，$a_{ij}(i>j)$ 全为 0，则构成的矩阵称为“上三角阵”，常记作 U。如果三角阵中的对角元都为 1，则称该三角阵为“单位上(下)三角阵。

若将一般方阵 A 的对角元构成的对角阵记作 C 的话，则 A 可记为

$$A=L+U-C$$

或

$$A=\begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}+\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} \\ 0 & 0 & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix}-\begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

3. 转置(阵)

设 A 为 m 行 n 列的矩阵(m 不一定等于 n)，矩阵 A 的元素 a_{ij} 与 $a_{ji}(i=1,2,\cdots,m,j=1,2,\cdots,n)$ 相互对换形成的矩阵，称为 A 的转置(阵)，记为 A^{T}

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix} \qquad A^{\mathrm{T}}=\begin{bmatrix} a_{11} & a_{21} & a_{31} & \cdots & a_{m1} \\ a_{12} & a_{22} & a_{32} & \cdots & a_{m2} \\ a_{13} & a_{23} & a_{33} & \cdots & a_{m3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & a_{3n} & \cdots & a_{mn} \end{bmatrix}$$

4. 逆(阵)

设 A 为方阵，若存在矩阵 B，使得 $A\times B=B\times A=I$(单位阵)，称 B 为 A 的逆(阵)，记作 A^{-1}，即 $A^{-1}A=AA^{-1}=I$

5. 对称阵

设 A 为方阵，其中 $a_{ij}=a_{ji}(i,j=1,2,\ldots,n,i\neq j)$，即元素关于对角元对称相等，则矩阵 A 称为“对称阵”。

6. 奇异阵

设 A 为方阵，若其行列式为 0，则称 A 为“奇异阵”，否则称为“非奇异阵”。当且仅当，A 阵至少有一个特征值为 0 时，A 为奇异阵。

7. 对称正定阵

设 A 为方阵，且对称，则其所有顺序主子式都大于 0，称 A 为“对称正定阵”。当且仅当，A 对称，且所有特征值都大于 0 时，A 为对称正定阵。并可推知，A 的所有对角元都大于 0；A 中

元素绝对值最大者为对角元。

5.2 模板类 matrix 及头文件

5.2.1 模板类 matrix 与矩阵运算

在模板类 matrix 中包含了有关定义矩阵对象的构造函数，以及有关矩阵运算的多种重载运算符，这些都在 matrix.h 中。另外还定义了有关矩阵运算的多种操作方法(函数)，都包含在相关文件 matrix.inl 中，该文件放在本书所附带的光盘中。

有关矩阵运算的操作方法(函数)包括：

1. 矩阵乘法；
2. 输出矩阵(逐行输出全矩阵)；
3. 输出矩阵(输出指定的某行)；
4. 矩阵转置；
5. 判断矩阵对称；
6. 判断矩阵对称正定；
7. 用全选主元法求矩阵行列式；
8. 用全选主元高斯(Gauss)消元法求一般矩阵的秩；
9. 用全选主元高斯－约当(Gauss-Jordan)消去法计算实(复)矩阵的逆矩阵；
10. 用“变量循环重新编号法”法求对称正定矩阵逆；
11. 用特兰持(Trench)法求托伯利兹(Toeplitz)矩阵逆；
12. 实矩阵 LU 分解；
13. 用豪斯荷尔德(Householder)变换对一般 $m\times n$ 阶的实矩阵进行 QR 分解；
14. 对称正定阵的乔里斯基(Cholesky)分解及求其行列式值；
15. 一般实矩阵的奇异值分解；
16. 广义逆的奇异值分解。

5.2.2 matrix.h

```
// Matrix.h    矩阵模板类头文件
// Ver 1.0.0.0
// 版权所有(C) 何渝，2002
// 最后修改：2002.5.31

#ifndef _MATRIX_H                    //避免多次编译
#define _MATRIX_H

#include <valarray>                  //模板类 valarray 的标准头文件
#include <comm.h>                    //公共头文件
#include <math.h>                    //数学头文件
#include <complex>                   //模板类 complex 头文件
```

```
//template <class _Ty = float>
template <class _Ty>
class matrix                        //矩阵类 matrix
{
typedef matrix<_Ty> _Myt;
private:
std::valarray<_Ty> m_Datas;  //定义一维数组对象 m_Datas
size_t m_stRow;                     //矩阵行数变量
size_t m_stCol;                     //矩阵列数变量
public:
typedef _Ty value_type;
//构造函数一(参数 1，2 分别为矩阵的行与列数)
/* * * * * *
矩阵类 matrix 的构造函数一
    构造函数中出现的 m_Datas 为 valarray 类的对象，申请 stRow * stCol 个单元，
单元内没赋值。对数组对象 m_Datas 使用了 valarray 类的构造函数
          explicit valarray(size_t n)
对私有变量 m_stRow 和 m_stCol 分别赋初值 stRow，stCol。
* * * * * */
matrix(size_t stRow, size_t stCol)
      : m_Datas(stRow * stCol),
        m_stRow(stRow), m_stCol(stCol)
{
      m_Datas.resize(GetRowNum() * GetColNum(), _Ty(0));
      /*
    构造函数中出现的 m_Datas 为 valarray 类的对象，若在函数体内调用的 resize(size_t n, const T&
c = T())函数有两个参数(参阅 valarray 中的定义)，第一个参数申请有矩阵行 * 列那么多元素个数的
存储空间，第二个参数对这些申请的空间赋予具有模式 T 的初值 0。如果不调用该函数，则缺省情况下
m_Datas长度为 0；另外，调用该函数但不使用第二个参数，则不会对 m_Datas 的任何一个元素赋初值。
   */
}

//构造函数二(参数 1 为指向矩阵的指针，参数 2，3 为矩阵行与列数)
/* * * * * *
矩阵类 matrix 的构造函数二
    对私有变量 m_stRow 和 m_stCol 分别赋初值 stRow，stCol。
    对数组对象 m_Datas 使用了 valarray 类的构造函数
              valarray(const _Ty * p, size_t n)
  m_Datas 初始化的第一个参数为矩阵 rhs 指针，第二个参数为 rhs 的元素总个数，
  即 rhs 行数 * 列数
* * * * * */
matrix(const _Ty* rhs, size_t stRow, size_t stCol)
      : m_Datas(rhs, stRow * stCol),
        m_stRow(stRow), m_stCol(stCol)
```

```
{
}

//构造函数三(拷贝构造函数,参数为对矩阵 matrix 的引用)
/* * * * * *
矩阵类 matrix 的构造函数三
    用引用矩阵 rhs 的数组对象 m_Datas 初始化 matrix 所定义对象的 m_Datas,用引用矩阵 rhs 的行数
rhs.GetRowNum()和列数 rhs.GetColNum()分别初始化私有变量 m_stRow 和 m_stCol
* * * * * */
matrix(const _Myt& rhs)
      : m_Datas(rhs.m_Datas),
        m_stRow(rhs.GetRowNum()), m_stCol(rhs.GetColNum())
{
}

size_t GetRowNum() const          //返回矩阵行数的函数
{
      return m_stRow;
}

size_t GetColNum() const          //返回矩阵列数的函数
{
      return m_stCol;
}

/* * * * *
    重载运算符(),获取二维矩阵某元素在一维数组 m_Datas 的位置(matrix 类定义的矩阵实际上已经
被转换为一维数组 m_Datas)
    当数组元素(i, j)在运算符左边(被写)时,使用_Ty& operator ()
    当数组元素(i, j)在运算符右边(被读)时,使用_Ty operator ()
* * * * */
_Ty& operator () (size_t stRow, size_t stCol)
{
Assert(stRow < GetRowNum());   //断定 stRow 不超实际矩阵行值
Assert(stCol < GetColNum());   //断定 stCol 不超实际矩阵列值

return m_Datas[stRow * GetColNum() + stCol];
}
/* * * * * *
    重载运算符(),返回二维矩阵中元素(stRow, stCol)在一维数组中的位置 stRow 与 stCol 分别为矩
阵某元素行与列的位置数
    当矩阵元素在运算符右边(被读)时使用
* * * * * */
const _Ty operator () (size_t stRow, size_t stCol) const
```

```
{
    Assert(stRow < GetRowNum());                    //断定 stRow 不超实际矩阵行值
    Assert(stCol < GetColNum());                    //断定 stCol 不超实际矩阵列值

    return m_Datas[stRow * GetColNum() + stCol];
}

// 赋值操作符
//矩阵与矩阵的自反 *，+，-运算符
_Myt& operator += (const _Myt& rhs)                 //矩阵与矩阵的自反 +
{
    Assert(GetRowNum() == rhs.GetRowNum());         //断定两矩阵的行数相等
    Assert(GetColNum() == rhs.GetColNum());         //断定两矩阵的列数相等
    //利用类 valarray 定义，使其对象 m_Datas 定义进行两矩阵相加
    m_Datas += rhs.m_Datas;
    //结果放在左边矩阵中，返回指向左边矩阵的指针
    return *this;
}

_Myt& operator -= (const _Myt& rhs)                 //矩阵与矩阵的自反 -
{
    Assert(GetRowNum() == rhs.GetRowNum());
    Assert(GetColNum() == rhs.GetColNum());
    //利用类 valarray 定义，使其对象 m_Datas 定义进行两矩阵相减
    m_Datas -= rhs.m_Datas;
    //结果放在左边矩阵中，返回指向左边矩阵的指针
    return *this;
}

_Myt& operator *= (const _Myt& rhs)                 //矩阵与矩阵的自反 *
{
    MatrixMultiply(*this, *this, rhs);              //调用矩阵乘法函数

    return *this;                                   //最后结果放在左边矩阵中
}

//矩阵自反加、减、乘以、除以数
_Myt& operator += (const _Ty& rhs)                  //矩阵自加数
{
    m_Datas += rhs;                                 //利用数组对象对每个元素加数

    return *this;                                   //结果放在原矩阵(数组 m_Datas)中
}
```

```
    _Myt& operator -= (const _Ty& rhs)                    //矩阵自减数
    {
        m_Datas -= rhs;

        return *this;
    }

    _Myt& operator *= (const _Ty& rhs)                    //矩阵自乘数
    {
        m_Datas *= rhs;

        return *this;
    }

    _Myt& operator /= (const _Ty& rhs)                    //矩阵自除以数
    {
        m_Datas /= rhs;

        return *this;
    }

    // 一元操作符 对矩阵(每个元素)赋予 + 或 - 号
    _Myt operator + () const                              //赋 + 号
    {
        return *this;                                     //不用作任何处理,维持原状
    }
    _Myt operator - () const                              //赋 - 号
    {
        _Myt mat(*this);
        mat.m_Datas = -mat.m_Datas;                       //每个元素赋 - 号

        return mat;
    }

    // 二元操作符
    //矩阵加数 mat = lhs + rhs
    friend _Myt operator + (const _Myt& lhs, const _Ty& rhs)
    {
        _Myt mat(lhs);                                    //新生成一新矩阵对象 mat
        mat.m_Datas += rhs;                               //对新矩阵对象每个元素加数

        return mat;                                       //返回新矩阵对象
    }
    //矩阵减数 mat = lhs - rhs
```

```
friend _Myt operator - (const _Myt& lhs, const _Ty& rhs)
{
    _Myt mat(lhs);
    mat.m_Datas - = rhs;

    return mat;
}

//矩阵乘以数 mat = lhs * rhs
friend _Myt operator * (const _Myt& lhs, const _Ty& rhs)
{
    _Myt mat(lhs);                                  //新生成一新矩阵对象 mat
    mat.m_Datas * = rhs;                            //对新矩阵对象每个元素乘以数

    return mat;                                     //返回新矩阵对象
}

//矩阵除以数 mat = lhs / rhs
friend _Myt operator / (const _Myt& lhs, const _Ty& rhs)
{
    _Myt mat(lhs);
    mat.m_Datas / = rhs;

    return mat;
}

//数加矩阵 mat = lhs + rhs
friend _Myt operator + (const _Ty& lhs, const _Myt& rhs)
{
    _Myt mat(rhs);                                  //新生成一新矩阵对象 mat
    mat.m_Datas + = lhs;                            //数加上新矩阵对象的每个元素

    return mat;
}

//数减矩阵 mat = lhs - rhs
friend _Myt operator - (const _Ty& lhs, const _Myt& rhs)
{
    _Myt mat(rhs);                                  //新生成一新矩阵对象 mat
    mat.m_Datas - = lhs;                            //数减新矩阵对象的每个元素

    return mat;
}
```

```
//数乘以矩阵 mat = lhs * rhs
friend _Myt operator * (const _Ty& lhs, const _Myt& rhs)
{
    _Myt mat(rhs);                                  //新生成一新矩阵对象 mat
    mat.m_Datas * = lhs;                            //对新矩阵对象每个元素乘以数

    return mat;
}

//矩阵加矩阵 mat = lhs + rhs
friend _Myt operator + (const _Myt& lhs, const _Myt& rhs)
{
    _Myt mat(lhs);                                  //新生成一新矩阵对象 mat，用左边阵初始化
    mat.m_Datas + = rhs.m_Datas;                    //加上右边矩阵对象每个相应元素

    return mat;                                     //返回新矩阵对象
}

//矩阵减矩阵 mat = lhs - rhs
friend _Myt operator - (const _Myt& lhs, const _Myt& rhs)
{
    _Myt mat(lhs);                                  //新生成一新矩阵对象 mat，用左边阵初始化
    mat.m_Datas - = rhs.m_Datas;                    //减去右边矩阵对象每个相应元素

    return mat;                                     //返回新矩阵对象
}

//矩阵乘以矩阵     mTmp = lhs * rhs
friend _Myt operator * (const _Myt& lhs, const _Myt& rhs)
{   //生成一个矩阵新对象 mTmp
    _Myt mTmp(lhs.GetRowNum(), rhs.GetColNum());    //没初始化

    MatrixMultiply(mTmp, lhs, rhs);                 //调用矩阵乘法函数

    return mTmp;                                    //用新矩阵对象返回
}

//矩阵比较
//比较两矩阵是否不相等,若相等返回 true,否则返回 false
friend bool operator ! = (const _Myt& lhs, const _Myt& rhs)
{
    if (lhs.GetRowNum() ! = rhs.GetRowNum())
      return true;
    if (lhs.GetColNum() ! = rhs.GetColNum())
```

```
            return true;

        for (size_t i = 0; i < lhs.m_Datas.size(); i + + )
        {
                if (lhs.m_Datas[i] ! = rhs.m_Datas[i])
                  return true;
        }
                  return false;
    }

    //比较两矩阵是否相等,若相等返回 true,否则返回 false
    friend bool operator = = (const _Myt& lhs, const _Myt& rhs)
    {
        if (lhs.GetRowNum() ! = rhs.GetRowNum())
          return false;
        if (lhs.GetColNum() ! = rhs.GetColNum())
          return false;

        for (size_t i = 0; i < lhs.m_Datas.size(); i + + )
        {
            if (lhs.m_Datas[i] ! = rhs.m_Datas[i])
              return false;
        }

        return true;
      }
    };
    typedef matrix<float> matrixf;
    typedef matrix<double> matrixd;
    typedef matrix<long double> matrixld;

    //矩阵乘法函数
    template <class _Tyout, class _Tylhs, class _Tyrhs>          //最后结果在 mOut 中
    matrix<_Tyout>& MatrixMultiply(matrix<_Tyout>& mOut, const matrix<_Tylhs>& lhs, const ma-
trix<_Tyrhs>& rhs);

    //输出矩阵          逐行输出全矩阵
    template <class _Ty>
    void MatrixLinePrint(const matrix<_Ty>& mOut);

    //输出矩阵          输出指定的某行
    template <class _Ty>
    void MatrixLinePrint(const matrix<_Ty>& mOut, size_t LineNo);
```

```
//矩阵转置          原阵在 mIn,转置后的矩阵在 mOut
template <class _Ty>
void MatrixTranspose(matrix<_Ty>& mIn, matrix<_Ty>& mOut);

//判断矩阵对称
template <class _Ty>                                    //对称返回 true,否则返回 false
bool MatrixSymmetry(const matrix<_Ty>& rhs);

//判断矩阵对称正定
template <class _Ty>
int MatrixSymmetryRegular(const matrix<_Ty>& rhs, int sym);

//用全选主元法求矩阵行列式
template <class _Ty>                                    //返回值为行列式值
long double MatrixDeterminant(const matrix<_Ty>& rhs);

//用全选主元高斯(Gauss)消元法求一般矩阵的秩
template <class _Ty>                                    //返回值为秩数
size_t MatrixRank(const matrix<_Ty>& rhs);

//用全选主元高斯—约当(Gauss-Jordan)消元法计算实(复)矩阵的逆矩阵
//矩阵类型必须是浮点型
template <class _Ty>
int MatrixInversionGS(matrix<_Ty>& rhs);

//用"变量循环重新编号法"法求对称正定矩阵逆
//矩阵类型必须是浮点型
template <class _Ty>
int MatrixSymmetryRegularInversion(matrix<_Ty>& rhs);

//用特兰持(Trench)法求托伯利兹(Toeplitz)矩阵逆
//矩阵类型必须是浮点型
template <class _Ty>
int MatrixToeplitzInversionTrench(const valarray<_Ty>& t,
                                  const valarray<_Ty>& tt, matrix<_Ty>& rhs);
//实矩阵 LU 分解
//矩阵类型必须是浮点型
template <class _Ty>
int MatrixLU(const matrix<_Ty>& rhs, matrix<_Ty>& lhs, matrix<_Ty>& uhs);

//用豪斯荷尔德(Householder)变换对一般 m * n 阶的实矩阵进行 QR 分解
//矩阵类型必须是浮点型
template <class _Ty>
int MatrixQR(matrix<_Ty>& rhs, matrix<_Ty>& rhq);
```

```
//对称正定阵的乔里斯基(Cholesky)分解及求其行列式值
//矩阵类型必须是浮点型
template <class _Ty>
long double MatrixSymmetryRegularCholesky(const matrix<_Ty>& rhs);

//一般实矩阵的奇异值分解
//矩阵类型必须是浮点型
template <class _Ty>
int MatrixSingularValue(matrix<_Ty>& a, matrix<_Ty>& u,
                                        matrix<_Ty>& v, _Ty eps);

//广义逆的奇异值分解
//矩阵类型必须是浮点型
template <class _Ty>
int GeneralizedInversionSingularValue(matrix<_Ty>& a, matrix<_Ty>& aa,
                          _Ty eps, matrix<_Ty>& u, matrix<_Ty>& v);

#include "Matrix.inl"                    //类及相关函数的定义头文件

#endif// _MATRIX_H
```

5.2.3 矩阵模板类 matrix 说明

1. 构造函数

(1) 构造函数一

```
matrix(size_t stRow, size_t stCol)
    : m_Datas(stRow * stCol),
      m_stRow(stRow), m_stCol(stCol)
{
}
```

构造函数一中，参数 stRow 与 stCol 分别为矩阵的行数与列数，m_Datas 为 valarray 类的对象，初始化时申请 stRow * stCol 个元素，且没赋值。

(2) 构造函数二

```
matrix(const _Ty* rhs, size_t stRow, size_t stCol)
    : m_Datas(rhs, stRow * stCol),
      m_stRow(stRow), m_stCol(stCol)
{
}
```

构造函数二中，第一个参数为指向矩阵 rhs 的指针，第二、三个参数 stRow 与 stCol 分别为矩阵 rhs 的行数与列数。初始化矩阵对象时，生成 valarray 类对象 m_Datas，其第一个参数也指向矩阵 rhs，第二个参数为矩阵 rhs 的元素总数，并将矩阵 rhs 的行数值与列数值初始化私有变量 m_stRow 和 m_stCol。

(3) 构造函数三

```
matrix(const _Myt& rhs)
```

```
    : m_Datas(rhs.m_Datas),
      m_stRow(rhs.GetRowNum()), m_stCol(rhs.GetColNum())
{
}
```

构造函数三中，参数为引用矩阵对象 rhs，并用 rhs.m_Datas 初始化构造的矩阵对象的 m_Datas，且用 rhs 的 GetRowNum()和 GetColNum()分别初始化私有变量 m_stRow 和 m_stCol。

2. 取矩阵行数函数

```
size_t GetRowNum() const
{
    return m_stRow;
}
```

该函数返回矩阵对象的行数，返回值为非负整数类型 size_t。

3. 取函数列数函数

```
size_t GetColNum() const
{
    return m_stCol;
}
```

该函数返回矩阵对象的列数，返回值为非负整数类型 size_t。

4. 运算符()

(1) 左()

```
_Ty& operator () (size_t stRow, size_t stCol)
{
      Assert(stRow < GetRowNum());              //断定 stRow 不超实际矩阵行值
      Assert(stCol < GetColNum());              //断定 stCol 不超实际矩阵列值
      return m_Datas[stRow * GetColNum() + stCol];
}
```

该运算符在另一运算符的左边时生效，返回矩阵元素所在的位置。

(2) 右()

```
const _Ty operator () (size_t stRow, size_t stCol)
{
      Assert(stRow < GetRowNum());              //断定 stRow 不超实际矩阵行值
      Assert(stCol < GetColNum());              //断定 stCol 不超实际矩阵列值
      return m_Datas[stRow * GetColNum() + stCol];
}
```

该运算符在另一运算符的右边时生效，返回矩阵元素所在的位置。

5. 矩阵与矩阵算术复合运算符

(1) +=

```
_Myt& operator += (const _Myt& rhs)             //矩阵与矩阵的自反+
{
      Assert(GetRowNum() == rhs.GetRowNum());//断定两矩阵的行数相等
      Assert(GetColNum() == rhs.GetColNum());//断定两矩阵的列数相等
      //利用类 valarray 定义，使其对象 m_Datas 定义进行两矩阵相加
```

```
        m_Datas += rhs.m_Datas;
        return *this;
}
```

矩阵与矩阵相加，结果存放在左边矩阵中。

(2) -=

```
_Myt& operator -= (const _Myt& rhs)                    //矩阵与矩阵的自反-
{
      Assert(GetRowNum() == rhs.GetRowNum()); //断定两矩阵的行数相等
      Assert(GetColNum() == rhs.GetColNum()); //断定两矩阵的列数相等
      //利用类 valarray 定义，使其对象 m_Datas 定义进行两矩阵相减
      m_Datas -= rhs.m_Datas;
      return *this;
}
```

矩阵与矩阵相减，结果存放在左边矩阵中。

(3) *=

```
_Myt& operator *= (const _Myt& rhs)                    //矩阵与矩阵的自反*
{
      MatrixMultiply(*this, *this, rhs);               //调用矩阵乘法函数
      return *this;                                    //最后结果放在左边矩阵中
}
```

矩阵与矩阵相乘，结果存放在左边矩阵中。

6. 矩阵与数复合算术运算符

(1) +=

```
_Myt& operator += (const _Ty& rhs)
{
      m_Datas += rhs;
      return *this;
}
```

矩阵每个元素加上数 rhs，结果存放在原矩阵中。

(2) -=

```
_Myt& operator -= (const _Ty& rhs)
{
      m_Datas -= rhs;
      return *this;
}
```

矩阵每个元素减去数 rhs，结果存放在原矩阵中。

(3) *=

```
_Myt& operator *= (const _Ty& rhs)
{
      m_Datas *= rhs;
      return *this;
}
```

矩阵每个元素乘以数 rhs,结果存放在原矩阵中。

(4) /=

```
_Myt& operator /= (const _Ty& rhs)
{
      m_Datas /= rhs;
      return *this;
}
```

矩阵每个元素除以数 rhs,结果存放在原矩阵中。

7. 符号运算符

(1) 十号

```
_Myt operator + () const                          //赋+号
{
return *this;                                     //不用作任何处理,维持原状
}
```

矩阵每个元素赋十号,结果在原矩阵中。

(2) 一号

```
_Myt operator - () const                          //赋-号
{
_Myt mat(*this);
mat.m_Datas = -mat.m_Datas;                       //每个元素赋-号
return mat;
}
```

矩阵每个元素赋一号,结果在原矩阵中。

8. 矩阵与数算术运算符

(1) +

```
friend _Myt operator + (const _Myt& lhs, const _Ty& rhs)
{ // mat = lhs + rhs
_Myt mat(lhs);                                    //生成一新矩阵对象 mat
mat.m_Datas += rhs;
return mat;                                       //返回新矩阵对象
}
```

矩阵 lhs 每个元素加上数 rhs,结果存放在新矩阵 mat 中并返回。

(2) -

```
friend _Myt operator - (const _Myt& lhs, const _Ty& rhs)
{ // mat = lhs - rhs
_Myt mat(lhs);                                    //生成一新矩阵对象 mat
mat.m_Datas -= rhs;
return mat;                                       //返回新矩阵对象
}
```

矩阵 lhs 每个元素减去数 rhs,结果存放在新矩阵 mat 中并返回。

(3) *

```
friend _Myt operator * (const _Myt& lhs, const _Ty& rhs)
```

```
{ // mat = lhs * rhs
_Myt mat(lhs);                                    //生成一新矩阵对象 mat
mat.m_Datas *= rhs;
return mat;                                       //返回新矩阵对象
}
```

矩阵 lhs 每个元素乘以数 rhs,结果存放在新矩阵 mat 中并返回。

(4) /

```
friend _Myt operator / (const _Myt& lhs, const _Ty& rhs)
{ // mat = lhs / rhs
_Myt mat(lhs);                                    //生成一新矩阵对象 mat
mat.m_Datas /= rhs;
return mat;                                       //返回新矩阵对象
}
```

矩阵 lhs 每个元素除以数 rhs,结果存放在新矩阵 mat 中并返回。

9. 数与矩阵算术运算符

(1) +

```
friend _Myt operator + (const _Ty& lhs, const _Myt& rhs)
{ // mat = lhs + rhs
_Myt mat(rhs);                                    //生成一新矩阵对象 mat
mat.m_Datas += lhs;
return mat;                                       //返回新矩阵对象
}
```

数 lhs 加上矩阵 rhs 的每个元素,结果存放在新矩阵 mat 中并返回。

(2) -

```
friend _Myt operator - (const _Ty& lhs, const _Myt& rhs)
{ // mat = lhs - rhs
_Myt mat(rhs);                                    //生成一新矩阵对象 mat
mat.m_Datas -= lhs;
return mat;                                       //返回新矩阵对象
}
```

数 lhs 减去矩阵 rhs 的每个元素,结果存放在新矩阵 mat 中并返回。

(3) *

```
friend _Myt operator * (const _Ty& lhs, const _Myt& rhs)
{ // mat = lhs * rhs
_Myt mat(rhs);                                    //生成一新矩阵对象 mat
mat.m_Datas *= lhs;
return mat;                                       //返回新矩阵对象
}
```

数 lhs 乘以矩阵 rhs 的每个元素,结果存放在新矩阵 mat 中并返回。

(4) /

```
friend _Myt operator / (const _Ty& lhs, const _Myt& rhs)
{ // mat = lhs / rhs
```

```
_Myt mat(rhs);                                    //生成一新矩阵对象 mat
mat.m_Datas /= lhs;
return mat;                                       //返回新矩阵对象
}
```

数 lhs 除以矩阵 rhs 的每个元素，结果存放在新矩阵 mat 中并返回。

10. 矩阵与矩阵算术运算符

(1) +

```
friend _Myt operator + (const _Myt& lhs, const _Myt& rhs)
{
_Myt mat(lhs);                                    //生成一新矩阵对象 mat
mat.m_Datas += rhs.m_Datas;                       //加上右边矩阵对象每个相应元素
return mat;                                       //返回新矩阵对象
}
```

矩阵 lhs 每个元素加上矩阵 rhs 的对应元素，结果存放在新矩阵 mat 中并返回。

(2) -

```
friend _Myt operator - (const _Myt& lhs, const _Myt& rhs)
{
_Myt mat(lhs);                                    //生成一新矩阵对象 mat
mat.m_Datas -= rhs.m_Datas;                       //减去右边矩阵对象每个相应元素
return mat;                                       //返回新矩阵对象
}
```

矩阵 lhs 每个元素减去矩阵 rhs 的对应元素，结果存放在新矩阵 mat 中并返回。

(3) *

```
friend _Myt operator * (const _Myt& lhs, const _Myt& rhs)
{
_Myt mTmp(lhs.GetRowNum(), rhs.GetColNum());      //生成一新矩阵对象 mTmp
MatrixMultiply(mTmp, lhs, rhs);                   //调用矩阵乘法函数
return mTmp;                                      //用新矩阵对象返回
}
```

矩阵 lhs 乘以矩阵 rhs，生成新矩阵各元素，结果存放在新矩阵 mat 中并返回。

11. 矩阵比较运算符

(1) !=

```
friend bool operator != (const _Myt& lhs, const _Myt& rhs)
{
if (lhs.GetRowNum() != rhs.GetRowNum())           //判断两矩阵行是否不相等
  return true;
if (lhs.GetColNum() != rhs.GetColNum())           //判断两矩阵列是否不相等
  return true;
for (size_t i = 0; i < lhs.m_Datas.size(); i++)
{
      if (lhs.m_Datas[i] != rhs.m_Datas[i])       //两矩阵对应元素是否不相等
        return true;
```

```
}
return false;
}
```

比较两矩阵是否不相等，若不相等返回 true，否则返回 false。

(2) ==

```
friend bool operator ! = (const _Myt& lhs, const _Myt& rhs)
{
if (lhs.GetRowNum() ! = rhs.GetRowNum())        //判断两矩阵行是否不相等
   return false;
if (lhs.GetColNum() ! = rhs.GetColNum())        //判断两矩阵列是否不相等
   return false;
for (size_t i = 0; i < lhs.m_Datas.size(); i + +)
{
      if (lhs.m_Datas[i] ! = rhs.m_Datas[i])   //两矩阵对应元素是否不相等
            return false;
}
return true;
}
```

比较两矩阵是否相等，若相等返回 true，否则返回 false。

5.2.4 矩阵基本运算示例

示例给出了如何使用作者定义的矩阵模板类 Matrix，如何利用该矩阵模板类所提供的各种方法(函数)，实现所需要的功能，获得所需要的结果。这当中包括定义矩阵对象，给新定义的矩阵对象赋值，矩阵对象的输出，矩阵与矩阵的加、减、乘运算，判断矩阵与矩阵是否相等，矩阵与数的相互加、减、乘、除等多种运算，这些运算不仅适用于实数矩阵，同样也适用于复数矩阵。

示例程序如下：

```
//MatrixExample.cpp 矩阵类中各种方法的应用示例

# include <iostream>                    //输入输出流
# include "Matrix.h"                    //矩阵类头文件
# include <complex>                     //复数模板类头文件
using namespace std;                    //名字空间

void main()                             // 定义控制台应用程序的入口点
{
//使用构造函数—定义两矩阵对象
matrix<int> matS(2, 1);                 //定义整型 2 * 1 矩阵 matS
matrix<int> matR(1, 2);                 //定义整型 1 * 2 矩阵 matR


//数组元素赋值
matS(0, 0) = 1;
matS(1, 0) = 2;
```

```
matR(0, 0) = 3;
matR(0, 1) = 4;

//输出两矩阵
cout << endl << "matS : " << endl;
MatrixLinePrint(matS);                                    //按行输出矩阵 matS
cout << endl << "matR : " << endl;
MatrixLinePrint(matR);                                    //按行输出矩阵 matR

const double dma[4][4] = {{ 1.2, 2.6, 3.7, 4.8 },
                          { 5.3, 6.0, 7.0, 8.0 },
                          { 9.4, 10.0, 11.6, 12.0 },
                          { 3.5, 14.0, 15.0, 16.8 }};
const double dmb[4][4] = {{ 3.0, -3.0, -2.0, 4.0 },
                          { 5.0, -5.0, 1.0, 8.0 },
                          { 1.0, 8.0, 5.0, -7.0 },
                          { 5.0, -1.0, -3.0, -1.0 } };

//使用构造函数二定义两矩阵对象 matA, matB
const matrix<double> matA(&dma[0][0], 4, 4);
const matrix<double> matB(&dmb[0][0], 4, 4);

//使用构造函数三定义两矩阵对象 matC
matrix<double> matC(matA);

//输出两矩阵 matA, matB
cout << endl << "matA : " << endl;
MatrixLinePrint(matA);                                    //按行输出矩阵 matA

cout << endl << "matB : " << endl;
MatrixLinePrint(matB);                                    //按行输出矩阵 matB

cout << endl << "matC : " << endl;
MatrixLinePrint(matC);                                    //按行输出矩阵 matC

cout << endl << "Row of matA : " << matA.GetRowNum();     //matA 行数
cout<<" \t Column of matA : "<<matA.GetRowNum()<<endl;    //matA 列数

matC += matA;                                             //矩阵与矩阵自 +
cout << endl << "matC += matA : " << endl;
MatrixLinePrint(matC);                                    //按行输出矩阵 matC

matC *= matA;                                             //矩阵自乘矩阵
cout << endl << "matC *= matA : " << endl;
```

```
MatrixLinePrint(matC);                                    //按行输出矩阵 matC

matC -= 12.3;                                             //矩阵自减数
cout << endl << "matC -= 12.3 : " << endl;
MatrixLinePrint(matC);                                    //按行输出矩阵 matC

matC = -matC;                                             //矩阵赋-号
cout << endl << "matC = -matC : " << endl;
MatrixLinePrint(matC);                                    //按行输出矩阵 matC

matC = matA * matB;                                       //矩阵乘矩阵
cout << endl << "matC = matA * matB : " << endl;
MatrixLinePrint(matC);                                    //按行输出矩阵 matC

matC = matC / 2.0;                                        //矩阵除以数
cout << endl << "matC = matC / 2.0 : " << endl;
MatrixLinePrint(matC);                                    //按行输出矩阵 matC

matC = 0.5 * matC;                                        //数乘以矩阵
cout << endl << "matC = 0.5 * matC : " << endl;
MatrixLinePrint(matC);                                    //按行输出矩阵 matC

matC = matA + matC;                                       //矩阵加矩阵
cout << endl << "matC = matA + matC : " << endl;
MatrixLinePrint(matC);                                    //按行输出矩阵 matC

//比较 matA 与 matC 两矩阵是否相同
cout << endl << "if (matA == matC) : " << (matA == matC);

//比较 matA 与 matB 两矩阵是否不相同
cout << endl << "if (matA != matB) : " << (matA != matB) << endl;

matrix< complex<float> > clhs(2, 1);                      //定义复数浮点型 2*1 数组 clhs
matrix< complex<float> > crhs(1, 2);                      //定义复数浮点型 1*2 数组 crhs

//给复数型矩阵元素赋值
clhs(0, 0) = complex<float> (1, 2);
clhs(1, 0) = complex<float> (3, 4);
crhs(0, 0) = complex<float> (5, 6);
crhs(0, 1) = complex<float> (7, 8);

cout << endl << "ComplexMatrix clhs :" << endl;
MatrixLinePrint(clhs);                                    //输出复矩阵 clhs
```

```
cout << endl << "ComplexMatrix crhs :" << endl;
MatrixLinePrint(crhs);                                    //输出复矩阵 crhs

matrix< complex<float> > cmm(clhs * crhs);                //cmm = clhs * crhs
cout << endl << "ComplexMatrix * ComplexMatrix :" << endl;
MatrixLinePrint(cmm);                                     //输出复矩阵 cmm = clhs * crhs

matrix< complex<float> > cmp(cmm);                        //生成 cmp,由 cmm 初始化
cmp += cmm;                                               //cmp += cmm
cout << endl << "ComplexMatrix * ComplexMatrix :" << endl;
MatrixLinePrint(cmp);                                     //输出复矩阵 cmp += cmm
}
```

矩阵基本运算示例结果如下：

```
matS :
1
2

matR :
3 4

matA :
1.2  2.6  3.7   4.8
5.3   6    7     8
9.4   10   11.6  12
3.5   14   15    16.8

matB :
3   -3   -2   4
5   -5   1    8
1   8    5    -7
5   -1   -3   -1

matC :
1.2  2.6  3.7   4.8
5.3   6    7     8
9.4   10   11.6  12
3.5   14   15    16.8

Row of matA : 4 Column of matA : 4

matC += matA :
 2.4   5.2   7.4   9.6
10.6   12    14    16
18.8   20    23.2  24
 7     28    30    33.6
```

```
matC * = matA ;
 133.6   245.84  275.12   303.2
263.92   463.56  525.62   583.68
430.64   736.88  838.68   931.84
 556.4    956.6  1073.9  1182.08

matC - = 12.3 ;
 121.3   233.54  262.82   290.9
251.62   451.26  513.32   571.38
418.34   724.58  826.38   919.54
 544.1    944.3  1061.6  1169.78

matC = -matC ;
 -121.3   -233.54  -262.82   -290.9
-251.62   -451.26  -513.32   -571.38
-418.34   -724.58  -826.38   -919.54
 -544.1    -944.3  -1061.6  -1169.78

matC = matA * matB ;
 44.3   8.2   4.3   -5.1
 92.9   2.1   6.4   12.2
149.8   2.6  13.2   24.4
179.5  22.7  31.6    4.2

matC = matC / 2.0 ;
22.15   4.1   2.15  -2.55
46.45  1.05   3.2    6.1
 74.9   1.3   6.6   12.2
89.75 11.35  15.8    2.1

matC = 0.5 * matC ;
11.075   2.05   1.075  -1.275
23.225  0.525   1.6     3.05
 37.45   0.65   3.3     6.1
44.875  5.675   7.9     1.05

matC = matA + matC ;
12.275   4.65   4.775  3.525
28.525  6.525   8.6    11.05
 46.85  10.65   14.9    18.1
48.375 19.675   22.9   17.85

if (matA = = matC) ; 0
```

```
if (matA ! = matB) ; 1

ComplexMatrix clhs ;
(1,2)
(3,4)

ComplexMatrix crhs ;
(5,6) (7,8)

ComplexMatrix * ComplexMatrix ;
( - 7,16) ( - 9,22)
( - 9,38) ( - 11,52)

ComplexMatrix * ComplexMatrix ;
( - 14,32) ( - 18,44)
( - 18,76) ( - 22,104)
```

5.3 矩阵转置与矩阵判别

5.3.1 矩阵转置

1. 功能

对矩阵进行转置操作，生成一新矩阵，新矩阵是老矩阵的转置矩阵。

2. 算法

设矩阵 A 为

$$A = (a_{ij})_{m\times n}, i = 1,2...,m;\ j = 1,2,...,n$$

生成一新矩阵 B，使 $b_{ij} = a_{ji}$，$i=1,2...,m;\ j=1,2,...,n$，即 $B=A^{\mathrm{T}}$。

3. 函数原型

```
//矩阵转置
template <class _Ty>
void MatrixTranspose(matrix<_Ty>& mIn, matrix<_Ty>& mOut);
```

4. 参数

mIn：matrix 类引用，要被转置的矩阵。

mOut：matrix 类引用，新生成的转置矩阵。

5. 返回值

无返回值。

6. 函数说明

需要进行转置的矩阵不一定是方阵，MatrixTranspose()函数对任何行与列的矩阵都可以进行处理。

7. 示例

对下面的 5×3 矩阵 A 进行转置运算，然后生成一新的 3×5 转置矩阵 B，接着对两矩阵进

行输出。

$$A = \begin{bmatrix} 1.01 & 1.02 & 1.03 \\ 2.04 & 2.05 & 2.06 \\ 3.07 & 3.08 & 3.09 \\ 4.01 & 4.04 & 4.08 \\ 5.02 & 5.05 & 5.07 \end{bmatrix}$$

示例程序如下：

```
//MatrixTranspose.cpp 矩阵转置与矩阵输出

#include <iostream>                                    //输入输出流
#include "Comm.h"
#include "Matrix.h"                                    //矩阵类及相关函数等的定义
using namespace std;                                   //名字空间

void main()                                            // 定义控制台应用程序的入口
点
{
double a[5][3] =
{
      {1.01, 1.02, 1.03},
      {2.04, 2.05, 2.06},
      {3.07, 3.08, 3.09},
      {4.01, 4.04, 4.08},
      {5.02, 5.05, 5.07}
};
matrix<double> ma(&a[0][0],5,3);
cout << "Matrix ma is : " << endl;
MatrixLinePrint(ma);                    //输出矩阵 ma
matrix<double> mb(3,5);                 //生成一新矩阵,为存放 ma 转置
MatrixTranspose(ma, mb);                //生成 ma 和转置阵 mb
int sR = mb.GetRowNum();                //取 mb 的行数
cout << endl << "Matrix mb is : " << endl;
for(int i = 0; i<sR; i++)
    MatrixLinePrint(mb, i);             //一行一行输出矩阵 mb
}
```

8. 示例结果

```
Matrix ma is :
        1.01   1.02   1.03
        2.04   2.05   2.06
        3.07   3.08   3.09
        4.01   4.04   4.08
        5.02   5.05   5.07
```

```
Matrix mb is :
        1.01   2.04   3.07   4.01   5.02
        1.02   2.05   3.08   4.04   5.05
        1.03   2.06   3.09   4.08   5.07
```

9. 示例说明

示例中调用了 MatrixLinePrint()函数，它是专为输出矩阵而设计的。其原型如下：

```
//输出矩阵          一行一行输出全矩阵
template <class _Ty>
void MatrixLinePrint(const matrix<_Ty>& mOut);

//输出矩阵          输出指定的某行
template <class _Ty>
void MatrixLinePrint(const matrix<_Ty>& mOut, size_t LineNo);
```

详述见 Matrix.h 和 Matrix.inl 文件。

5.3.2 对称矩阵

1. 功能

判别一矩阵是否为对称矩阵。

2. 算法

设矩阵 A 为

$$(a_{ij})_{m\times n}, i = 1,2,\ldots,m;\ j = 1,2,\ldots,n$$

若 $m=n$，且 $a_{ij}=a_{ji}, i=1,2,\ldots,m;\ j=1,2,\ldots,n$，矩阵 A 为对称矩阵，即 $A=A^{\mathrm{T}}$。

3. 函数原型

```
//判断矩阵对称
template <class _Ty>
bool MatrixSymmetry(const matrix<_Ty>& rhs);
```

4. 参数

rhs：matrix 类引用，引用要进行判别的矩阵。

5. 返回值

bool 类型。

true：矩阵为对称矩阵。

false：矩阵不为对称矩阵。

6. 示例

示例对下述三个矩阵进行是否为对称阵的判别：

$$A=\begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\end{bmatrix}\qquad B=\begin{bmatrix}3 & -3 & -2\\ -3 & 8 & 4\\ -2 & 4 & -3\end{bmatrix}\qquad C=\begin{bmatrix}3 & -3 & -2\\ -2 & 4 & -3\end{bmatrix}$$

示例程序如下：

```
//MatrixSymmetry.cpp 判断矩阵对称

#include <iostream>                    //输入输出流
#include "Matrix.h"                    //矩阵类及相关函数等的定义
using namespace std;                   //名字空间

void main()                            // 定义控制台应用程序的入口点
{
const double dma[4][4] =
{
      { 1.0, 2.0, 3.0, 4.0 },
      { 5.0, 6.0, 7.0, 8.0 },
      { 9.0, 10.0, 11.0,12.0 },
      { 13.0,14.0, 15.0,16.0 }
};
const int dmb[3][3] =
{
      { 3, -3, -2 },
      { -3, 8,  4 },
      { -2, 4, -3 }
};

const double dmc[2][3] =
{
      { 3.0,   -3.0, -2.0 },
      { -2.0,   4.0, -3.0 }
};
const matrix<double> matA(&dma[0][0], 4, 4);
const matrix<double> matB(&dmb[0][0], 3, 3);
const matrix<double> matC(&dmc[0][0], 2, 3);
if(MatrixSymmetry(matA))
      cout << "The matA is Symmetry." << endl;
else
      cout << "The matA is not Symmetry." << endl;

if(MatrixSymmetry(matB))
      cout << "The matB is Symmetry." << endl;
else
      cout << "The matB is not Symmetry." << endl;

if(MatrixSymmetry(matC))
      cout << "The matC is Symmetry." << endl;
else
      cout << "The matC is not Symmetry." << endl;
```

```
}
```

7. 示例结果

```
The matA is not Symmetry.
The matB is Symmetry.
The matC is not Symmetry.
```

5.3.3 正定(对称)矩阵

1. 功能

(1) 判别矩阵是否为正定矩阵。

(2) 除使用(1)判别矩阵是否为正定矩阵外,还可以判别矩阵是否为对称矩阵。

2. 算法

(1) 矩阵 A 为 n 阶方阵,A 的所有顺序主子式都大于 0,则矩阵 A 是正定矩阵。对 $k=1,2,\ldots,n$,对矩阵 A 的 k 阶主子式逐个计算,若都大于 0,则矩阵 A 为正定矩阵。

(2) 矩阵 A 为方阵,且对称,A 的所有顺序主子式都大于 0,则矩阵 A 是对称正定矩阵。若 A 为对称正定矩阵,则其所有对角元都大子 0。

判别矩阵是否为正定矩阵,使用了全选主元高斯(Gauss)消元法计算矩阵行列式其算法参阅本章 3.4.1 计算矩阵行列式一节中的介绍。程序在判别矩阵是否正定的过程中,调用了本章 3.4.1 中函数 MatrixDeterminant();程序在判别矩阵是否为对称矩阵的过程中,调用本章 5.3.2 中函数 MatrixSymmetry()。

3. 函数原型

```
//判断矩阵对称正定
template <class _Ty>
int MatrixSymmetryRegular(const matrix<_Ty>& rhs, int sym);
```

4. 参数

rhs:matrix 类引用,要进行判别的矩阵。

sym:int 类型,若还要判别矩阵是否对称,赋值 1,否则不判别矩阵是否为对称矩阵,赋值。

5. 返回值

int 类型。

1: 正定。

2: 对称正定。

−1: 不是方阵。

−2: 不是对称阵。

−3: 对角元不大于 0。

0: 某顺序主子式不大于 0。

6. 函数说明

矩阵 A 可以是既正定,又对称。但矩阵 A 是正定阵时,不一定是对称矩阵;反之,矩阵 A 是对称矩阵时,也不一定是正定矩阵。

函数 MatrixSymmetryRegular()判别矩阵是否是正定矩阵,或者是否是对称且正定矩阵。若已知矩阵是对称阵,或者只需要判别矩阵是否为正定矩阵,可将函数 MatrixSymmetryReg-

ular()的第二个参数值设为0。

若要判别矩阵是否为半正定矩阵(顺序主子式≥0),或者是否为负定矩阵(顺序主子式<0),或者半负定矩阵(顺序主子式≤0),则只需对程序作少许修改。

函数 MatrixSymmetryRegular()在运行过程中若已经计算了矩阵的顺序主子式,会显示出其主子式的数值,如在示例中对矩阵 $\boldsymbol{B}$ 与 $\boldsymbol{C}$ 的判别。

7. 示例

示例对下述三个矩阵进行对称正定矩阵的判别:

$$\boldsymbol{A}=\begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\end{bmatrix} \qquad \boldsymbol{B}=\begin{bmatrix}3 & -3 & -2\\ -3 & 8 & 4\\ -2 & 4 & -3\end{bmatrix} \qquad \boldsymbol{C}=\begin{bmatrix}5 & 7 & 6 & 5\\ 7 & 10 & 8 & 7\\ 6 & 8 & 10 & -9\\ 5 & 7 & 9 & 10\end{bmatrix}$$

示例程序如下:

```
// MatrixSymmetryRegular.cpp 判断对称正定阵
#include <iostream>                          //输入输出流
#include "Matrix.h"                          //矩阵类及相关函数等的定义
using namespace std;                         //名字空间

void main()                                  // 定义控制台应用程序的入口点
{
void DislayMatrixInfo(int sValue);
int returnValue;

const double dma[4][4] =
{
      { 1.0,  2.0,  3.0,  4.0 },
      { 5.0,  6.0,  7.0,  8.0 },
      { 9.0, 10.0, 11.0, 12.0 },
      { 13.0,14.0, 15.0, 16.0 }
};

const double dmb[3][3] =
{
      { 3.0,   -3.0, -2.0 },
      { -3.0,   8.0,  4.0 },
      { -2.0,   4.0,  3.0 }
};

const double dmc[4][4] =
{
      { 5.0,  7.0,  6.0,   5.0},
      { 7.0, 10.0,  8.0,   7.0},
      { 6.0,  8.0, 10.0, -9.0},
```

```
        { 5.0, 7.0, 9.0, 10.0}
};
const matrix<double> matA(&dma[0][0], 4, 4);
const matrix<double> matB(&dmb[0][0], 3, 3);
const matrix<double> matC(&dmc[0][0], 4, 4);

cout << "matA : " << endl;
int sym(1);                                    //判别矩阵是否为对称阵
returnValue = MatrixSymmetryRegular(matA, sym);
DislayMatrixInfo(returnValue);

cout << endl << "matB : " << endl ;
sym = 1;                                       //判别矩阵是否为对称阵
returnValue = MatrixSymmetryRegular(matB, sym);
DislayMatrixInfo(returnValue);

cout << endl << "matC : " << endl;
sym = 0;                                       //不判别矩阵是否为对称阵
returnValue = MatrixSymmetryRegular(matC, sym);
DislayMatrixInfo(returnValue);
}

void DislayMatrixInfo(int sValue)
{
switch(sValue)
{
  case 1: cout << "The matrix is regular." << endl;
          break;
  case 2: cout << "The matrix is symmetry and regular." << endl;
          break;
  case -1: cout << "The matrix is not a square." << endl;
           break;
  case -2: cout << "The matrix is not symmetry." << endl;
           break;
  case -3: cout << "The matrix is not regular." << endl;
           break;
  case 0: cout << "The matrix is not regular." << endl;;
          break;
}
}
```

8. 示例结果

```
matA :
The matrix is not symmetry.
```

```
matB :
K = 1 Determinant = 3
K = 2 Determinant = 15
K = 3 Determinant = 13
The matrix is symmetry and regular.

matC :
K = 1  Determinant = 5
K = 2  Determinant = 1
K = 3  Determinant = 2
K = 4  Determinant = 55
The matrix is regular.
```

5.4 矩阵行列式及矩阵的秩

5.4.1 矩阵行列式

1. 功能

用全选主元高斯(Gauss)消元法计算 n 阶方阵 A 所对应的行列式的值。

2. 算法

高斯消元法的实质就是将一个 n 阶方阵 A 通过一系列变化，变换成一个上三角阵，称为“消元”过程。最后得到的三角阵的对角元的乘积值就是矩阵 A 的行列式的值。

设矩阵 A 为 n 阶方阵

$$A=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}_{n\times n}$$

为保证数值计算的稳定性，采用全选主元法。对于 k 从 1 到 $n-1$，分为三步。

(1) 从矩阵 A 的第 k 行、第 k 列开始的右下角子阵中选取绝对值最大的元素作为该次消元的主元，并通过行交换与列交换将它交换到主元素的位置上(这种行或列的交换是初等变换，不影响矩阵行列式的值)。当 k 为 0 时，A 阵为原始阵，即还未作任何变换。下图 A 阵中的元素 $a_{kk}^{(k)}$ 就是在第 k 次选主元后，经行或(和)列交换后放在主元的位置上的。

$$A=\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1k}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2k}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)} \end{bmatrix}$$

(2) 归一。令

$$a_{kj}=a_{kj}/a_{kk},j=k+1,\ldots,n$$

(3) 消元。令

$$a_{ij} = a_{ij} - a_{ik} \times a_{kj}, i,j = k+1,\ldots,n$$

也即将主元 $a_{kk}^{(k)}$ 所在列下方的诸元素消为 0，对主元 $a_{kk}^{(k)}$ 所在行下方的其余元素作变换。

若计算正常结束，矩阵将变换成一个上三角阵，三角阵的对角元的乘积就是该矩阵的行列式值。在计算的第 k 步过程中，若碰到主元为 0(这时元素 $a_{ij}, i,j=k,\ldots,n$，将全为 0)，表示该矩阵的行列式为 0，即该矩阵是奇异阵。

3. 函数原型

```
//全选主元法求矩阵行列式
template <class _Ty>
long double MatrixDeterminant(const matrix<_Ty>& rhs);
```

4. 参数

rhs：matrix 类引用，引用要进行计算行列式的矩阵。

5. 返回值

long double 类型。

0：表示所计算的矩阵不是方阵就是奇异阵。

非 0：程序正常结束，此值为矩阵行列式的数值。

6. 示例

示例分别计算下述两个矩阵的行列式值：

$$\boldsymbol{A}=\begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\end{bmatrix} \quad \boldsymbol{B}=\begin{bmatrix}3.8 & -3.0 & -2.0 & 4.0\\ 5.0 & -5.2 & 1.0 & 8.0\\ 11.0 & 8.0 & 5.6 & -7.0\\ 5.0 & -1.0 & -3.0 & -1.4\end{bmatrix} \quad \boldsymbol{C}=\begin{bmatrix}-1 & 2 & 3\\ 9 & 0 & 2\\ 5 & 1 & 6\end{bmatrix}$$

示例程序如下：

```
//MatrixDeterminant.cpp 高斯全选主元法矩阵求行列式

#include <iostream>                    //输入输出流
#include "Matrix.h"                    //矩阵类及相关函数等的定义
using namespace std;                   //名字空间

void main()                            // 定义控制台应用程序的入口点
{
const double dma[4][4] =
{
      { 1.0, 2.0, 3.0, 4.0 },
      { 5.0, 6.0, 7.0, 8.0 },
      { 9.0, 10.0,11.0,12.0 },
      { 13.0,14.0,15.0,16.0 }
};
const double dmb[4][4] =
{
      { 3.8, -3.0, -2.0, 4.0 },
```

```
        { 5.0, -5.2, 1.0,   8.0 },
        { 11.0, 8.0,  5.6,   -7.0 },
        { 5.0, -1.0, -3.0, -1.4}
};

const matrix<double> matA(&dma[0][0], 4, 4);
const matrix<double> matB(&dmb[0][0], 4, 4);
cout << "Det(matA) = " << MatrixDeterminant(matA) << endl;
cout << endl << "Det(matB) = " << MatrixDeterminant(matB) << endl;

const int imc[3][3] =
{
    {-1, 2, 3},
    { 9,  0, 2},
    { 5,  1, 6}
};

const matrix<int> matC(&imc[0][0], 3, 3);
cout << endl << "Det(matC) = " << MatrixDeterminant(matC) << endl;
}
```

7. 示例结果

```
Det(matB) = 0
Det(matA) = 759.238
Det(marC) = -59
```

5.4.2 矩阵秩

1. 功能

用全选主元高斯(Gauss)消元法计算一般矩阵的秩。

2. 算法

设有 $m\times n$ 矩阵

$$\boldsymbol{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

取 $k=\min\{m,n\}$。对于 $r=1,2,\ldots,k$,用全选主元高斯消元法,将 $\boldsymbol{A}$ 变换为上三角阵,直到某一步 $a_{rr}=0$ 为止,矩阵 $\boldsymbol{A}$ 的秩为 r。

3. 函数原型

```
//全选主元高斯(Gauss)消元法求一般矩阵的秩
template <class _Ty>
size_t MatrixRank(const matrix<_Ty>& rhs);
```

4. 参数

rhs:matrix 类引用,引用要进行求秩的矩阵。

5. 返回值

非负整数类型。返回值为矩阵的秩值。

6. 函数说明

MatrixRank()函数对所处理的矩阵为一般矩阵，不一定为方阵，见示例中对矩阵 C 的运算。

7. 示例

示例分别计算下述三个矩阵的秩：

$$A=\begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\end{bmatrix} \qquad B=\begin{bmatrix}3 & -3 & -2\\ -3 & 8 & 4\\ -2 & 4 & -3\end{bmatrix} \qquad C=\begin{bmatrix}-1 & 2 & 3 & 4\\ 5 & -6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\\ 17 & 18 & 19 & 20\end{bmatrix}$$

示例程序如下：

```
// MatrixRank.cpp 用全选主元高斯消元法求一般矩阵的秩

#include <iostream>                          //输入输出流
#include "Matrix.h"                          //矩阵类及相关函数等的定义
using namespace std;                         //名字空间

void main()                                  // 定义控制台应用程序的入口点
{
size_t mRank;

const double dma[4][4] = {{ 1.0,2.0,  3.0,  4.0 },
                          { 5.0, 6.0,  7.0,  8.0 },
                          { 9.0, 10.0, 11.0, 12.0 },
                          { 13.0,14.0, 15.0, 16.0 } };

const double dmb[3][3] = {{ 3.0,     -3.0, -2.0 },
                          { -3.0,  8.0,   4.0 },
                          { -2.0,  4.0,    -3.0 } };

const double dmc[5][4] = {{ -1.0, 2.0,   3.0,  4.0},
                          { 5.0,    -6.0, 7.0,  8.0},
                          { 9.0,    10.0, 11.0, 12.0},
                          { 13.0,   14.0, 15.0, 16.0},
                          { 17.0,   18.0, 19.0, 20.0} };

const matrix<double> matA(&dma[0][0], 4, 4);
const matrix<double> matB(&dmb[0][0], 3, 3);
const matrix<double> matC(&dmc[0][0], 5, 4);
```

```
mRank = MatrixRank(matA);
cout << endl << "The rank of matA is : " << mRank << endl;
mRank = MatrixRank(matB);
cout << endl << "The rank of matB is : " << mRank << endl;
mRank = MatrixRank(matC);
cout << endl << "The rank of matC is : " << mRank << endl;
}
```

8. 示例结果

```
The rank of matA is : 2
The rank of matA is : 3
The rank of matA is : 4
```

5.5 逆 矩 阵

5.5.1 高斯—约当法求逆矩阵

1. 功能

用全选主元高斯—约当(Gauss-Jordan)消元法计算实(复)矩阵的逆矩阵 $\boldsymbol{A}^{-1}$。

2. 算法

用全选主元高斯—约当法求矩阵的逆矩阵的步骤如下。

首先,对于 k 从 1 到 n 作如下操作。

(1) 从第 k 行,第 k 列开始的右下角子阵中选取绝对值最大的元素,并记住此元素所在的行号和列号,再通过行交换与列交换将它交换到主元位置上。这一步称为"全选主元"。

(2) $a_{kk}=1/a_{kk}$

(3) $a_{kj}=a_{kj}a_{kk}$, $j=1,2,\ldots,n$; $j\neq k$

(4) $a_{ij}=a_{ij}-a_{ik}a_{kj}$, $i,j=1,2,\ldots,n$; $i,j\neq k$

(5) $a_{ik}=-a_{ik}a_{kk}$, $i,j=1,2,\ldots,n$; $i\neq k$

最后,根据在全选主元的过程中所记录的行、列交换的信息进行恢复,恢复的原则如下:

在全选主元过程中,先交换的行和列后进行恢复;原来的行(列)交换用列(行)交换来恢复。

该法与全选主元高斯消元法的区别在于:在消元的过程中,不仅要对主元下方的元素作变换,也要同时对主元的上方元素作变换。

3. 函数原型

```
//用全选主元高斯—约当(Gauss-Jordan)消元法计算实(复)矩阵的逆矩阵
template <class _Ty>
int MatrixInversionGS(matrix<_Ty>& rhs);
```

4. 参数

rhs:matrix 类引用,要进行求逆的矩阵 $\boldsymbol{A}$,正常结束后其逆仍存放于其中。注意:其数据类型必须是浮点类型!

5. 返回值

0:表示矩阵不是方阵,或者矩阵是奇异阵,计算中止。

1:表示程序运行正常结束,矩阵的逆存放在原矩阵中。

6. 函数说明

MatrixInversionGS()函数所处理的矩阵可以是实数矩阵，也可以是复数矩阵。若是复数矩阵，返回的逆矩阵也是复数矩阵。参阅示例中对矩阵 C 的运算。

特别要注意的是，对求逆矩阵的数据进行输入时，一定要用浮点数类型，因为在求逆的过程中，数据的类型可能发生变化。例如，当原矩阵的数据类型为整型时，计算出的逆矩阵的数据类型不一定是整型。

7. 示例

示例对实矩阵 A 进行求逆 A^{-1} 计算，并计算了 $A^{-1}A$，以验证其是否等于单位阵 I。A 矩阵如下：

$$A=\begin{bmatrix}0.2368 & 0.2471 & 0.2568 & 1.2671\\ 1.1161 & 0.1254 & 0.1397 & 0.1490\\ 0.1582 & 1.1675 & 0.1768 & 0.1871\\ 0.1968 & 0.2071 & 1.2168 & 0.2271\end{bmatrix}$$

示例同时也对复数矩阵 C 进行求逆 C^{-1} 计算，并计算了 $C^{-1}C$，以验证其是否等于复数单位阵 I。其中复数矩阵 C 为

$$C=\begin{bmatrix}(0.2368,0.1345) & (0.2471,0.1678) & (0.2568,0.1825) & (1.2671,1.1161)\\ (1.1161,1.2671) & (0.1254,0.2617) & (0.1397,0.7024) & (0.1490,0.2721)\\ (0.1582,-0.2836) & (1.1675,-1.1967) & (0.1768,0.3558) & (0.1871,-0.2078)\\ (0.1968,0.3567) & (0.2071,-1.2345) & (1.2168,2.1185) & (0.2271,0.4773)\end{bmatrix}$$

示例程序如下：

```
// MatrixInversionGS.cpp 全选主元高斯—约当(Gauss-Jordan)法求矩阵逆

#include <iostream>                    //输入输出流
#include "Matrix.h"                    //矩阵类及相关函数等的定义
using namespace std;                   //名字空间

void main()                            // 定义控制台应用程序的入口点
{
const double dma[4][4] =
{
      {0.2368,0.2471, 0.2568,1.2671},
      {1.1161,0.1254, 0.1397,0.1490},
      {0.1582,1.1675, 0.1768,0.1871},
      {0.1968,0.2071, 1.2168,0.2271}
};

matrix<double> matA(&dma[0][0], 4, 4);
matrix<double> matB(matA);

cout << "matA : " << endl;
```

```
MatrixLinePrint(matA);                          //按行输出矩阵 matA
if(MatrixInversionGS(matA)>0)                   //调用求逆函数,判断是否>0
{
  cout << endl << "Inversion(matA) : " << endl;
  MatrixLinePrint(matA);                        //按行输出矩阵 matA 的逆
  cout << endl << "Inversion(matA) * matA : " << endl;
  matA = matA * matB;
  MatrixLinePrint(matA);                        //按行输出矩阵 matA 与其逆的乘积
}

const complex<double> cdma[4][4] =
{
{
  complex<double>(0.2368,0.1345), complex<double>(0.2471,0.1678),
  complex<double>(0.2568,0.1825), complex<double>(1.2671,1.1161)
},
{
  complex<double>(1.1161,1.2671), complex<double>(0.1254,0.2617),
  complex<double>(0.1397,0.7024), complex<double>(0.1490,0.2721)
},
{
  complex<double>(0.1582,-0.2836), complex<double>(1.1675,-1.1967),
  complex<double>(0.1768,0.3558), complex<double>(0.1871,-0.2078)
},
{
  complex<double>(0.1968,0.3567), complex<double>(0.2071,-1.2345),
  complex<double>(1.2168,2.1185), complex<double>(0.2271,0.4773)
}
};
matrix<complex<double> > cmatA(&cdma[0][0], 4, 4);
matrix<complex<double> > cmatB(cmatA);

cout << endl << "cmatA : " << endl;
MatrixLinePrint(cmatA);                         //按行输出矩阵 cmatA
if(MatrixInversionGS(cmatA)>0)                  //判断返回值是否>0
{
  cout << endl << "Inversion(cmatA) : " << endl;
  MatrixLinePrint(cmatA);                       //按行输出矩阵 cmatA 的逆 cmaBT

  cout << endl << "Inversion(cmatA) * cmatA : " << endl;
  cmatA = cmatA * cmatB;
  MatrixLinePrint(cmatA);                       //按行输出矩阵 cmatA 与其逆的乘积
 }
}
```

8. 示例结果

```
matA :
    0.2368      0.2471      0.2568      1.2671
    1.1161      0.1254      0.1397       0.149
    0.1582      1.1675      0.1768      0.1871
    0.1968      0.2071      1.2168      0.2271

Inversion(matA) :
      -0.0859208        0.937944      -0.0684372      -0.0796077
       -0.10559      -0.0885243        0.905983      -0.0991908
       -0.127073       -0.111351       -0.116967        0.878425
        0.851606       -0.135456       -0.140183       -0.143807

Inversion(matA) * matA :
                   1     -2.42861e-017     -2.77556e-017     -3.1225e-017
         1.73472e-017                 1     -4.16334e-017      3.46945e-018
                   0     -2.77556e-017                 1     -2.77556e-017
      -2.08167e-017      1.73472e-017     -5.55112e-017                 1
cmatA :
      (0.2368,0.1345)   (0.2471,0.1678)   (0.2568,0.1825)   (1.2671,1.1161)
      (1.1161,1.2671)   (0.1254,0.2617)   (0.1397,0.7024)    (0.149,0.2721)
     (0.1582,-0.2836)  (1.1675,-1.1967)  (0.1768,0.3558)  (0.1871,-0.2078)
      (0.1968,0.3567)   (0.2071,-1.2345)  (1.2168,2.1185)   (0.2271,0.4773)
Inversion(cmatA) :
(-0.000166154,0.0437188)(0.490758,-0.484665)  (0.0205624,-0.269565)(-0.19001,0.127122)
(-0.0707871,0.117074)    (-0.0478357,0.149802)  (0.558269,0.518356)    (-0.0562662,-0.144273)
(-0.177702,0.103559)     (-0.143253,0.114798)  (0.0735632,0.459489)  (0.262187,-0.470442)
(0.483786,-0.443136)      (-0.032098,0.0412441)  (-0.126404,-0.1192)   (-0.000630771,0.0901346)

Inversion(cmatA) * cmatA :
(1,-2.08167e-017)(-2.77556e-017,-5.55112e-017)(-1.11022e-016,-5.55112e-017)(-5.55112e-017,-3.46945e-017)
(0,4.16334e-017)(1,0)                         (8.32667e-017,5.55112e-017) (6.93889e-017,4.16334e-017)
(0,0)           (0,1.11022e-016)              (1,-4.16334e-017)           (0,0)
(0,0)           (4.16334e-017,-2.08167e-017)(0,-2.77556e-017)             (1,-3.46945e-017)
```

9. 示例说明

示例中求出矩阵 matA 的逆矩阵仍存放于 matA 中，矩阵 matB 复制了原矩阵 matA。最后用 matA×matB(即 matA 的逆矩阵乘以 matA)验证其正确性，即 $matA^{-1} \times matA = I$(单位阵)。

示例也对复矩阵 cmatA 进行了求逆计算，结果也存放在 cmatA 中，cmatB 复制了原复矩阵 matA，最后也用 cmatA×cmatB(即 cmatA 的逆矩阵乘以 cmatA)验证其正确性，即 $cmatA^{-1} \times cmatA = CI$(复数单位阵)。

示例中单位阵 I 的非对角元素不是严格的 0 值，而是一些很小的接近 0 的数，这是电子数字计算机本身进行数值计算性质所决定的。复数单位阵中有的元素也不是严格的 0 值，原因

也同实单位阵一样。

5.5.2 对称正定矩阵的逆矩阵

1. 功能

用“变量循环重新编号法”计算对称正定矩阵 A 的逆矩阵 A^{-1}。

2. 算法

变量循环重新编号法的计算公式如下：

$$a_{nn} = 1/a_{11}$$

$$a_{nj} = -a_{1j}/a_{11}, \quad j = 2,3,\cdots,n$$

$$a_{in} = a_{i1}/a_{11}, \quad i = 2,3,\cdots,n$$

$$a_{ij} = a_{ij} - a_{i1}a_{1j}/a_{11}, \quad i,j = 2,3,\cdots,n$$

当 A 为对称正定阵时，其逆矩阵 A^{-1} 也为对称正定阵。

3. 函数原型

```
//用“变量循环重新编号法”法求对称正定矩阵逆
template <class _Ty>
int MatrixSymmetryRegularInversion(matrix<_Ty>& rhs);
```

4. 参数

rhs：matrix 类引用，要进行求逆的矩阵 A，正常结束后其逆仍存放于其中。

5. 返回值

≠2：表示矩阵不是对称正定阵，计算中止。

2：表示程序运行正常结束，矩阵的逆存放在原矩阵中。

6. 函数说明

注意：输入的矩阵数据类型应是浮点类型。

7. 示例

示例对下列矩阵 A 进行求逆 A^{-1} 计算，并计算 $A^{-1}A$，以验证其是否等于单位阵 I。

$$A = \begin{bmatrix} 5 & 7 & 6 & 5 \\ 7 & 10 & 8 & 7 \\ 6 & 8 & 10 & 9 \\ 5 & 7 & 9 & 10 \end{bmatrix}$$

示例程序如下：

```
//MatrixSymmetryRegularInversion.cpp 用“变量循环重新编号法”法求对称正定矩阵逆

#include <iostream>    //输入输出流
#include "Matrix.h"    //矩阵类及相关函数等的定义
using namespace std;   //名字空间

void main()            // 定义控制台应用程序的入口点
{
const double dma[4][4] =
{
```

```
        { 5.0, 7.0, 6.0, 5.0},
        { 7.0, 10.0, 8.0, 7.0},
        { 6.0, 8.0, 10.0, 9.0},
        { 5.0, 7.0, 9.0, 10.0}
    };

    matrix<double> matA(&dma[0][0], 4, 4);
    matrix<double> matB(matA);
    cout << "matA: " << endl;
    MatrixLinePrint(matA);                    //按行输出矩阵 matA

    int iValue = MatrixSymmetryRegularInversion(matA);
    if(iValue > 1)
    {
        cout << endl << "Inversion(matA) : " << endl;
        MatrixLinePrint(matA);                //按行输出矩阵 matA 的逆
        cout << endl << "Inversion(matA) * matA : " << endl;
        matA = matA * matB;
        MatrixLinePrint(matA);                //按行输出矩阵 matA 与逆的乘积
    }
    else
        cout << "matA 不是对称正定阵!" << endl;
    }
```

8. 示例结果

```
matA:
     5      7      6      5
     7      10     8      7
     6      8      10     9
     5      7      9      10
K = 1    Determinant = 5
K = 2    Determinant = 1
K = 3    Determinant = 2
K = 4    Determinant = 1

Inversion(matA) :
       68       -41       -17        10
      -41        25        10        -6
      -17        10         5        -3
       10        -6        -3         2
Inversion(matA) * matA :
                    1     5.68434e-014     5.68434e-014     1.42109e-014
    -1.77636e-014                    1    -5.68434e-014    -2.13163e-014
    -3.55271e-015    -1.77636e-014                    1    -7.10543e-015
     3.55271e-015     1.77636e-014     1.06581e-014                    1
```

9. 说明

示例中对称正定阵 matA 进行了求逆运算，逆矩阵仍存放在 matA 中，另用 matB 复制 matA 原矩阵，最后用 matA 的逆与原矩阵相乘进行了验证，看其是否为单位阵 I。所得结果的单位阵中非对角元应为 0 值，实际计算结果是一些很小的数，而非严格的 0 值，这是电子数字计算机进行数值计算本身性质所决定的。

程序中调用了函数 MatrixSymmetryRegular()对要求逆的矩阵进行了是否为对称正定阵的判别，请参阅本章 5.3.3 节。如果事先已经知道所要计算的矩阵肯定是对称正定矩阵，可以屏蔽掉函数 MatrixSymmetryRegular()，以节省时间，提高效率。

5.5.3 托伯利兹矩阵的逆矩阵

1. 功能

用特兰持(Trench)法计算托伯利兹(Tocplitz)矩阵 A 的逆矩阵 A^{-1}。

2. 算法

设 n 阶托伯利兹矩阵为

$$T^{(n)}=\begin{bmatrix} t_0 & t_1 & t_2 & \cdots & t_{n-1} \\ \tau_1 & t_0 & t_1 & \cdots & t_{n-2} \\ \tau_2 & \tau_1 & t_0 & \cdots & t_{n-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \tau_{n-1} & \tau_{n-2} & \tau_{n-3} & \cdots & t_0 \end{bmatrix}$$

简称 n 阶 T 型矩阵。其求逆过程如下。

取初值 $\alpha_0=t_0, c_0^{(0)}=\tau_1/t_0, r_0^{(0)}=t_1/t_0$

第一步　对于 k 从 0 到 $n-3$ 作如下工作

(1) $c_i^{(k+1)}=c_i^{(k)}+\dfrac{r_{k-1}^{(k)}}{\alpha_k}\left(\sum_{j=1}^{k+1}c_{k+1-j}^{(k)}\tau_j-\tau_{k+2}\right),\quad i=0,1,\cdots,k$

$$c_{k+1}^{(k+1)}=\frac{1}{\alpha_k}\left(\tau_{k+2}-\sum_{j=1}^{k+1}c_{k+1-j}^{(k)}\tau_j\right)$$

(2) $r_i^{(k+1)}=r_i^{(k)}+\dfrac{c_{k-1}^{(k)}}{\alpha_k}\left(\sum_{j=1}^{k+1}r_{k+1-j}^{(k)}t_j-t_{k+2}\right), i=0,1,\cdots,k$

$$r_{k+1}^{(k+1)}=\frac{1}{\alpha_k}\left(t_{k+2}-\sum_{j=1}^{k+1}r_{k+1-j}t_j\right)$$

(3) $\alpha_{k+1}=t_0-\sum_{j=1}^{k+2}t_jc_{j-1}^{(k+1)}$

最后可算出 α_{n-2} 以及 $c_i^{(n-2)}$ 和 $r_i^{(n-2)}$，$(i=0,1,\cdots,n-2)$。

第二步　计算逆矩阵 $B^{(n)}$ 中的各元素

$b_{00}=1/\alpha_{n-2}$

$b_{0,j+1}=-r_j^{(n-2)}/\alpha_{n-2}, j=0,1,\cdots,n-2$

$b_{i+1,0}=-c_i^{(n-2)}/\alpha_{n-2}, i=0,1,\cdots,n-2$

$$b_{i+1,j+1}=b_{ij}+\frac{1}{\alpha_{n-2}}\left[c_i^{(n-2)}r_j^{(n-2)}-r_{n-i-2}^{(n-2)}c_{n-j-2}^{(n-2)}\right], i,j=0,1,\cdots,n-2$$

3. 函数原型

```
//特兰持(Trench)法求托伯利兹(Toeplitz)矩阵逆
template <class _Ty>
int MatrixToeplitzInversionTrench(const valarray<_Ty>& t,
                                  const valarray<_Ty>& tt, matrix<_Ty>& rhs);
```

4. 参数

t:valarray 类型,存放 n 阶 T 型矩阵中的上三角元素 $t_0, t_1, \ldots, t_{n-1}$。

tou:valarray 类型,其中后 $n-1$ 个元素 tou[0],tou[1],...,tou[$n-1$]依次存放 n 阶 T 型矩阵中的元素 $\tau_1, \tau_2, \ldots, \tau_{n-1}$。

rhs:matrix 类引用,引用要计算的 n 阶 T 型矩阵。注意:其数据类型必须是浮点型。

5. 返回值

-1:矩阵不是方阵。

0:运算失败。

1:程序运行正常结束,其矩阵的逆存放在原矩阵中。

6. 函数说明

注意:所有输入参数的数据类型应是浮点类型。

7. 示例

计算 6 阶 T 型矩阵 $\boldsymbol{T}^{(6)}$ 的逆矩阵 $\boldsymbol{B}$,然后计算 $\boldsymbol{T}^{(6)}\boldsymbol{B}$ 是否等于单位阵 $\boldsymbol{I}$,以验证其结果的正确性。

$$\boldsymbol{T}^{(6)} = \begin{bmatrix} 10 & 5 & 4 & 3 & 2 & 1 \\ -1 & 10 & 5 & 4 & 3 & 2 \\ -2 & -1 & 10 & 5 & 4 & 3 \\ -3 & -2 & -1 & 10 & 5 & 4 \\ -4 & -3 & -2 & -1 & 10 & 5 \\ -5 & -4 & -3 & -2 & -1 & 10 \end{bmatrix}$$

其中

$$\boldsymbol{T} = [10,5,4,3,2,1]$$

$$\mathrm{TOU} = [0,-1,-2,-3,-4,-5]$$

程序如下:

```
//MatrixToeplitzInversionTrench.cpp 特兰持法求托伯利兹矩阵逆

#include <iostream>                    //输入输出流
#include "Matrix.h"                    //矩阵类及相关函数等的定义
using namespace std;                   //名字空间

void main()                            // 定义控制台应用程序的入口点
{
  const double t[6] = {10.0, 5.0, 4.0, 3.0, 2.0, 1.0};
  const double tuo[6] = { 0.0, -1.0, -2.0, -3.0, -4.0, -5.0};
  const double dma[6][6] = {{10.0, 5.0, 4.0, 3.0, 2.0, 1.0},
                            {-1.0,10.0, 5.0, 4.0, 3.0, 2.0},
                            {-2.0, -1.0,10.0, 5.0, 4.0, 3.0},
```

```
                                {-3.0,-2.0,-1.0,10.0, 5.0, 4.0},
                                {-4.0,-3.0,-2.0,-1.0,10.0, 5.0},
                                {-5.0,-4.0,-3.0,-2.0,-1.0,10.0} };
    matrix<double> matT(&dma[0][0], 6, 6);
    matrix<double> matB(matT);                          //用 matT 初始化 matB

    cout << endl << "matB = Inversion(matT) : " << endl;
    if(MatrixToeplitzInversionTrench(t, tuo, matB))     //现在 matB 是 matT 的逆
    MatrixLinePrint(matB);                              //按行输出矩阵 matB

    cout << endl << "matI = matT * matB : " << endl;
    matrix<double> matI(matT * matB);                   //matT * matB 得到单位阵 matI
    MatrixLinePrint(matI);                              //按行输出矩阵 matI
    }
```

8. 示例结果

matT :

10	5	4	3	2	1
-1	10	5	4	3	2
-2	-1	10	5	4	3
-3	-2	-1	10	5	4
-4	-3	-2	-1	10	5
-5	-4	-3	-2	-1	10

matB = Inversion(matT) :

0.0946884	-0.0469953	-0.0137211	-0.0017727	0.00190745	0.0038019
-0.00427534	0.0949232	-0.0469003	-0.0137122	-0.00181915	0.00190745
-0.000988143	-0.00473167	0.0948032	-0.0469175	-0.0137122	-0.0017727
0.0017727	-0.000988082	-0.00474397	0.0948032	-0.0469003	-0.0137211
0.0130644	0.00209908	-0.000988082	-0.00473167	0.0949232	-0.0469953
0.0469986	0.0130644	0.0017727	-0.000988143	-0.00427534	0.0946884

matI = matT * matB :

1	3.46741e-005	-0.00693481	-4.33681e-019	-5.11743e-017	-1.38778e-017
-0.00210127	1.00104	0.000339164	-0.00689547	-4.23289e-005	-8.43695e-005
0.00713412	-0.00560017	1.00002	0.000207183	-0.00675264	0.000240309
-0.00348283	0.00874343	-0.00512863	1.00008	0.000139531	-0.00688163
-8.66964e-005	-2.40994e-005	0.00970547	-0.00499818	1.00001	-0.000174668
5.55112e-017	-2.77556e-017	1.17961e-016	0.00970874	-0.005	1

9. 示例说明

示例用 dma 分别初始化矩阵对象 matT 和 matB，计算得到的逆存放于 matB 中，最后将 matT 与 matB 相乘，以验证其所计算得到的逆的正确性，看其是否为单位阵。

所得结果的单位阵中有的元素不是严格的数值 1 和 0，这是电子数字计算机进行数值计算本身性质所决定的。

5.6 矩阵分解

5.6.1 实矩阵的三角(*LU*)分解

1. 功能

对 n 阶实方阵进行三角(*LU*)分解，即矩阵 $A=LU$，其中 L 为单位下三角阵，U 为上三角阵，即

$$L=\begin{bmatrix}1 & 0 & 0 & \cdots & 0\\ l_{21} & 1 & 0 & \cdots & 0\\ l_{31} & l_{32} & 1 & \cdots & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1\end{bmatrix} \qquad U=\begin{bmatrix}u_{11} & u_{12} & u_{13} & \cdots & u_{1n}\\ 0 & u_{22} & u_{23} & \cdots & u_{2n}\\ 0 & 0 & u_{33} & \cdots & u_{3n}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & 0 & \cdots & u_{nn}\end{bmatrix}$$

2. 算法

令矩阵 Q 为

$$Q=L+U-I=\begin{bmatrix}u_{11} & u_{12} & u_{13} & \cdots & u_{1n}\\ l_{21} & u_{22} & u_{23} & \cdots & u_{2n}\\ l_{31} & l_{32} & u_{33} & \cdots & u_{3n}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ l_{n1} & l_{n2} & l_{n3} & \cdots & u_{nn}\end{bmatrix}$$

其计算步骤为：

对于 k 从 1 到 $n-1$

(1) $a_{ik}=a_{ik}/a_{kk}$，$i=k+1,\ldots,n$；

(2) $a_{ij}=a_{ij}-a_{ik}a_{kj}$，$i,j=k+1,\ldots,n$。

3. 函数原型

```
//实矩阵 LU 分解
template <class _Ty>
int MatrixLU(const matrix<_Ty>& rhs, matrix<_Ty>& lhs, matrix<_Ty>& uhs)
```

4. 参数

rhs：matrix 类引用，要进行分解的原矩阵 A。注意：其数据类型必须是浮点型。

lhs：matrix 类引用，分解后的单位下三角阵 L。

uhs：matrix 类引用，分解后的上三角阵 U。

5. 返回值

int 类型。

−1：表示矩阵不是方阵。

0：主元为 0。

1：表示程序运行正常结束，分解成功。

6. 函数说明

(1) 由于本算法没有使用选主元法,故数值计算不稳定。

(2) 程序中对要计算的矩阵进行了是否为方阵的判别,如果事先已经知道所要计算的矩阵肯定是方阵,可以屏蔽其判别,以节省时间,提高效率。

7. 示例

示例对下述三个矩阵进行 LU 分解:

$$A=\begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\end{bmatrix} \qquad B=\begin{bmatrix}3 & -3 & -2 & 2\\ -3 & 8 & 4 & 1\\ -2 & 4 & 6 & 3\\ 2 & 1 & 3 & 9\end{bmatrix} \qquad C=\begin{bmatrix}2 & 4 & 4 & 2\\ 3 & 3 & 12 & 6\\ 2 & 4 & -1 & 2\\ 4 & 2 & 1 & 1\end{bmatrix}$$

示例程序如下:

```
//MatrixLU.cpp LU分解及求其行列式值

#include <iostream>                    //输入输出流
#include "Matrix.h"                    //矩阵类及相关函数等的定义
using namespace std;                   //名字空间

void main()                            // 定义控制台应用程序的入口点
{
void DislayMatrixInfo(int iRet, matrix<double>& A,
                      matrix<double>& L, matrix<double>& U);
int iRet;

const double dma[4][4] = {{ 1.0, 2.0, 3.0, 4.0 },
                          { 5.0,  6.0, 7.0, 8.0 },
                          { 9.0,  10.0,11.0,12.0 },
                          { 13.0, 14.0,15.0,16.0 }};

const double dmb[4][4] = {{ 3.0, -3.0,-2.0,2.0 },
                          { -3.0,8.0,  4.0,  1.0 },
                          { -2.0,4.0,  6.0,  3.0 },
                          { 2.0,  1.0,  3.0,  9.0 } };

const double dmc[4][4] = {{2.0, 4.0, 4.0,  2.0},
                          {3.0,  3.0, 12.0, 6.0},
                          {2.0,  4.0, -1.0,2.0},
                          {4.0,  2.0, 1.0,  1.0} };

matrix<double> matA(&dma[0][0], 4, 4);
matrix<double> matB(&dmb[0][0], 4, 4);
matrix<double> matC(&dmc[0][0], 4, 4);

matrix<double> matA_L(4,4);
```

```
    matrix<double> matA_U(4,4);

    iRet = MatrixLU(matA, matA_L, matA_U);
    cout << "matA : " << endl ;
    DislayMatrixInfo(iRet, matA, matA_L, matA_U);

    iRet = MatrixLU(matB, matA_L, matA_U);
    cout << endl << "matB : " << endl ;
    DislayMatrixInfo(iRet, matB, matA_L, matA_U);

    iRet = MatrixLU(matC, matA_L, matA_U);
    cout << endl << "matC : " << endl;
    DislayMatrixInfo(iRet, matC, matA_L, matA_U);

}
void DislayMatrixInfo(int iRet, matrix<double>& A,
                      matrix<double>& L, matrix<double>& U)
{
if(iRet = = -1)
        cout << "矩阵不是方阵!" << endl;
else
    if(iRet = = 0)
        cout << "矩阵分解失败!" << endl;
    else
    {
        MatrixLinePrint(A);
        cout << endl << "L : " << endl;
        MatrixLinePrint(L);
        cout << endl << "U : " << endl;
        MatrixLinePrint(U);
    }
}
```

8. 示例结果

```
matA :
矩阵分解失败!

matB :
          3      -3      -2       2
         -3       8       4       1
         -2       4       6       3
          2       1       3       9
L :
```

```
          1          0          0          0
         -1          1          0          0
 -0.666667        0.4          1          0
  0.666667        0.6   0.810345          1
```

U：

```
          3         -3         -2          2
          0          5          2          3
          0          0    3.86667    3.13333
          0          0          0    3.32759
```

matC：

```
          2          4          4          2
          3          3         12          6
          2          4         -1          2
          4          2          1          1
```

L：

```
          1          0          0          0
        1.5          1          0          0
          1          0          1          0
          2          2        3.8          1
```

U：

```
          2          4          4          2
          0         -3          6          3
          0          0         -5          0
          0          0          0         -9
```

9. 示例说明

示例中的矩阵元素下标从 0 开始，注意与算法中标记的不一样。

5.6.2 一般实矩阵的 QR 分解

1. 功能

用豪斯荷尔德(Householder)变换对一般 $m\times n$ 阶的实矩阵进行 QR 分解。

2. 算法

设 $m\times n$ 的实矩阵 A 列线性无关，则可以将 A 分解为 $A=QR$ 的形式。其中 Q 为 $m\times m$ 的正交矩阵，R 为 $m\times n$ 的上三角形矩阵。

豪斯荷尔德方法对 A 进行 QR 分解的步骤如下。

令 $$s=\min(m-1,n),Q=I_{m\times n}。$$

对于 k 从 0 到 $s-1$ 作以下几步：

(1) 确定豪斯荷尔德变换

$$Q^{(k)}=\left[\begin{array}{c|c} I_k & 0 \\ \hline 0 & Q_{m-k} \end{array}\right]$$

其中

$$I_k=\underbrace{\left.\begin{bmatrix}1 & 0 & \cdots & 0\\ 0 & 1 & \cdots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \cdots & 1\end{bmatrix}\right\}k\text{ 列}}_{k\text{ 行}}$$

$$\tilde{Q}_{m-k}=\begin{bmatrix}(1-2u_k^2) & -2u_ku_{k+1} & \cdots & -2u_ku_{m-1}\\ -2u_{k+1}u_k & (1-2u_{k+1}^{2^*}) & \cdots & -2u_{k+1}u_{m-1}\\ \vdots & \vdots & \ddots & \vdots\\ -2u_{m-1}u_k & -2u_{m-1}u_{k+1} & \cdots & (1-2u_{m-1}^2)\end{bmatrix}$$

矩阵中 $u_j(j=k,k+1,\cdots,m-1)$ 的计算公式如下：

$$\eta=\max_{k\leqslant i\leqslant m-1}|a_{ik}|$$

$$\alpha=-\operatorname{sign}(a_{kk})\sqrt{\sum_{i=k}^{m-1}(a_{ik}/\eta)^2\cdot\eta}$$

$$\rho=\sqrt{2\alpha(\alpha-a_{kk})}$$

$$u_k=\frac{1}{\rho}(a_{kk}-\alpha)\Rightarrow a_{kk}$$

$$u_i=\frac{1}{\rho}a_{ik}\Rightarrow a_{ik},i=k+1,\cdots,m-1$$

(2) 用 $Q^{(k)}$ 左乘 Q，即 $Q^{(k)}Q\Rightarrow Q$

其计算公式如下：

$$\left.\begin{aligned}t&=\sum_{l=k}^{m-1}a_{lk}\cdot q_{lk}\\ q_{ij}-2a_{ik}t&\Rightarrow q_{ij}\end{aligned}\right\}\begin{aligned}&j=0,1,\cdots,m-1\\ &i=k,k+1,\cdots,m-1\end{aligned}$$

(3) 用 $Q^{(k)}$ 左乘 A，即 $Q^{(k)}A\Rightarrow A$ 。

其计算公式如下：

$$\left.\begin{aligned}t&=\sum_{l=k}^{m-1}a_{lk}\cdot a_{lj}\\ a_{ij}-2a_{ik}t&\Rightarrow a_{ij}\end{aligned}\right\}\begin{aligned}&j=k+1,\cdots,m-1\\ &i=k,\cdots,m-1\end{aligned}$$

$$a\Rightarrow a_{kk}$$

最后，矩阵 A 的右上三角部分即为 R，而矩阵 Q^{T} 为正交矩阵。

3. 函数原型

```
//用豪斯荷尔德(Householder)变换对一般 m * n 阶的实矩阵进行 QR 分解
template <class _Ty>
int MatrixQR(matrix<_Ty>& rhs, matrix<_Ty>& rhq);
```

4. 参数

rhs：matrix 类引用，存放要分解的原矩阵，程序正常结束后，存放分解后的 R 矩阵。

rhq：matrix 类引用，存放分解成功后的 Q 矩阵。

注意：rhs 和 rhq 的数据类型应为浮点类型。

5. 返回值

0:表示分解失败。

1:表示程序运行正常结束。分解的结果分别存放于 rhs 和 rhq 矩阵中。

6. 示例

示例对下列 4×3 的矩阵 A 进行了 QR 分解：

$$A = \begin{bmatrix} 1 & 1 & -1 \\ 2 & 1 & 0 \\ 1 & -1 & 0 \\ -1 & 2 & 1 \end{bmatrix}$$

示例程序如下：

```
//MatrixQR.cpp 用豪斯荷尔德(Householder)变换对一般 m * n 阶的实矩阵进行 QR 分解

# include <iostream>                    //输入输出流
# include "Matrix.h"                    //矩阵类及相关函数等的定义
using namespace std;                    //名字空间

void main()                             // 定义控制台应用程序的入口点
{
const double dma[4][3] =
{
    { 1.0, 1.0,    -1.0},
    { 2.0, 1.0,     0.0},
    { 1.0, -1.0,    0.0},
    {-1.0, 2.0,     1.0}
};

matrix<double> matA(&dma[0][0], 4, 3);//生成 A 阵
matrix<double> matQ(4, 4);            //定义存放 Q 阵的矩阵对象

cout << "matA :" << endl;
MatrixLinePrint(matA);                //按行输出矩阵 matA

cout << endl << "matQ :" << endl;
if(MatrixQR(matA, matQ))              //生成 Q 阵,且 R 阵在 matA 右上三角
MatrixLinePrint(matQ);                //按行输出矩阵 matQ

cout << endl << "matR :" << endl;
MatrixLinePrint(matA);                //按行输出矩阵 matR
}
```

7. 示例结果

```
matA :
```

```
        1          1         -1
        2          1          0
        1         -1          0
       -1          2          1
matQ :
   -0.377964  -0.377964   0.755929   0.377964
   -0.755929  -0.377964  -0.377964  -0.377964
   -0.377964   0.377964  -0.377964   0.755929
    0.377964  -0.755929  -0.377964   0.377964
matR :
   -2.64575   -2.22045e-016   0.755929
          0   -2.64575       -0.377964
          0    0             -1.13389
          0    0              0
```

8. 示例说明

示例运行结果中有的元素应为0值，实际计算结果是一些很小的数，而非严格的0值，这是电子数字计算机进行数值计算本身性质所决定的。

5.6.3 对称正定矩阵的乔里斯基分解及行列式值

1. 功能

用乔里斯基(Cholesky)分解法求对称正定矩阵的三角分解，并计算其矩阵行列式的值。

2. 算法

设 n 阶矩阵 $\boldsymbol{A}$ 为对称正定阵，则存在一个实的非奇异的下三角阵 $\boldsymbol{L}$，使 $\boldsymbol{A}=\boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}$，其中

$$\boldsymbol{L}=\begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix}$$

乔里斯基分解的步骤如下。

对于 $j=1,2,\ldots,n$

(1) $l_{jj}=(a_{jj}-\sum_{k=1}^{j}l_{jk}^2)^{\frac{1}{2}}$

(2) $l_{ij}=\dfrac{(a_{ij}-\sum_{k=1}^{j}l_{ik}l_{jk})}{l_{jj}},i=j+2,\cdots,n$

$\boldsymbol{A}$ 的行列式值为 $\det(A)=(\prod_{k=1}^{n}l_{kk})^2$

3. 函数原型

```
template <class _Ty>
long double MatrixSymmetryRegularCholesky(matrix<_Ty>& rhs);
```

4. 参数

rhs:matrix 类引用,为要分解的对称正定阵 A。

注意:矩阵类型应为浮点类型。

5. 返回值

long double 类型。

0:表示矩阵 A 不是对称正定阵。

>0:为矩阵的行列式值。

程序运行结束后,原矩阵中存放着分解后的结果。

6. 函数说明

分解正常结束后,原要分解的矩阵 A 右上三角部分即为 R,而矩阵 Q 为正交阵。

程序中调用了函数 MatrixSymmetryRegular(),对要求逆的矩阵进行了是否为对称正定阵的判别。如果事先已经知道所要计算的矩阵肯定是对称正定矩阵,可以屏蔽掉函数 MatrixSymmetryRegular(),以节省时间,提高效率。

7. 示例

示例对下述三个矩阵进行了乔里斯基分解及求行列式:

$$A=\begin{bmatrix}1&2&3&4\\5&6&7&8\\9&10&11&12\\13&14&15&16\end{bmatrix}\quad B=\begin{bmatrix}3&-3&-2\\-3&8&4\\-2&4&3\end{bmatrix}\quad C=\begin{bmatrix}5&7&6&5\\7&10&8&7\\6&8&10&9\\5&7&9&10\end{bmatrix}$$

示例程序如下:

```
//MatrixSymmetryRegularCholesky.cpp 对称正定阵的乔里斯基(Cholesky)分解及求其行列式值

#include <iostream>                    //输入输出流
#include "Matrix.h"                    //矩阵类及相关函数等的定义
using namespace std;                   //名字空间

void main()                            // 定义控制台应用程序的入口点
{
const double dma[4][4] = {{ 1.0,  2.0,  3.0, 4.0 },
                          { 5.0,  6.0,  7.0, 8.0 },
                          { 9.0,  10.0, 11.0,12.0 },
                          { 13.0, 14.0, 15.0,16.0 }};

const double dmb[3][3] = {{ 3.0,    -3.0, -2.0 },
                          { -3.0,  8.0,   4.0 },
                          { -2.0,  4.0,   3.0 } };

const double dmc[4][4] = {{ 5.0, 7.0,  6.0,  5.0},
                          { 7.0, 10.0, 8.0,  7.0},
                          { 6.0, 8.0,  10.0, 9.0},
                          { 5.0, 7.0,  9.0,  10.0} };
```

```
matrix<double> matA(&dma[0][0], 4, 4);
matrix<double> matB(&dmb[0][0], 3, 3);
matrix<double> matC(&dmc[0][0], 4, 4);

cout << "matA: " << endl;
long double mDet = MatrixSymmetryRegularCholesky(matA);
if(mDet > 0)
{
    cout << endl << "Determinant(matA) = " << mDet << endl;
    MatrixLinePrint(matA);
}
else cout << "The matA is not symmetry and regular." << endl;

cout << endl << "matB: " << endl;
mDet = MatrixSymmetryRegularCholesky(matB);
if(mDet > 0)
{
    cout << "Determinant(matB) = " << mDet << endl;
    MatrixLinePrint(matB);
}
else cout << "The matB is not symmetry and regular." << endl;

cout << endl << "matC: " << endl;
mDet = MatrixSymmetryRegularCholesky(matC);
if(mDet > 0)
{
    cout << "Determinant(matC) = " << mDet << endl;
    MatrixLinePrint(matC);
}
else cout << "The matC is not symmetry and regular." << endl;
}
```

8. 示例结果

```
matA:
The matA is not symmetry and regular.

matB:
  K = 1  Determinant = 3
  K = 2  Determinant = 15
  K = 3  Determinant = 13
Determinant(matB) = 13
      1.73205             0             0
     -1.73205       2.23607             0
      -1.1547      0.894427      0.930949
```

```
matC:
  K = 1  Determinant = 5
  K = 2  Determinant = 1
  K = 3  Determinant = 2
  K = 4  Determinant = 1
Determinant(matC) = 1
    2.23607              0          0          0
     3.1305       0.447214          0          0
    2.68328      -0.894427    1.41421          0
    2.23607              0    2.12132   0.707107
```

9. 示例说明

示例中矩阵元素下列从 0 开始，与算法介绍中不一样。

5.6.4 一般实矩阵的奇异值分解

1. 功能

用豪斯荷尔德(Householder)及变型 **QR** 算法对一般矩阵进行奇异值分解。

2. 算法

设 $\boldsymbol{A}$ 为 $m\times n$ 的实矩阵，则存在一个 $m\times m$ 的列正交矩阵 $\boldsymbol{U}$ 和 $m\times n$ 的列正交矩阵 $\boldsymbol{V}$，使

$$\boldsymbol{A}=\boldsymbol{U}\begin{bmatrix}\sum & 0\\ 0 & 0\end{bmatrix}\boldsymbol{V}^{\mathrm{T}}$$

成立，其中 $\sum=\mathrm{diag}(\sigma_0,\sigma_1,\cdots,\sigma_p)$，$(p\leqslant\min(m,n)-1)$ 且 $\sigma_0\geqslant\sigma_1\geqslant\cdots\geqslant\sigma_p>0$。

上式称为实矩阵 $\boldsymbol{A}$ 的奇异值分解式，$\sigma_i(i=0,1,\cdots,p)$ 称为 $\boldsymbol{A}$ 的奇异值。

利用 $\boldsymbol{A}$ 的奇异值分解式，可以计算 $\boldsymbol{A}$ 的广义逆 $\boldsymbol{A}^{+}$(见本章 5.6.5 节)。利用 $\boldsymbol{A}$ 的广义逆可以求解线性最小二乘问题(见第 7 章 7.2.12 节)。

奇异值分解分两大步。

第一步　用豪斯荷尔德变换将 $\boldsymbol{A}$ 化为双对角线矩阵

$$\boldsymbol{B}=\tilde{\boldsymbol{U}}^{\mathrm{T}}\boldsymbol{A}\tilde{\boldsymbol{V}}=\begin{bmatrix}s_0 & e_0 & & \\ & s_1 & e_1 & 0\\ & & \ddots & \ddots\\ & 0 & s_{p-1} & e_{p-1}\\ & & & s_p\end{bmatrix}$$

其中

$$\tilde{\boldsymbol{U}}=\boldsymbol{U}_0,\boldsymbol{U}_1,\cdots\boldsymbol{U}_{k-1},k=\min(n,m-1)$$

$$\tilde{\boldsymbol{V}}=\boldsymbol{V}_0,\boldsymbol{V}_1,\cdots\boldsymbol{V}_{l-1},l=\min(m,n-2)$$

$\tilde{\boldsymbol{U}}$ 中的每一个变换 $\boldsymbol{U}_j(j=0,1,\cdots,k-1)$ 将 $\boldsymbol{A}$ 中第 j 列主对角线以下的元素变为零；而 $\tilde{\boldsymbol{V}}$ 中的每一个变换 $\boldsymbol{V}_j(j=0,1,\cdots,l-1)$ 将 $\boldsymbol{A}$ 中第 j 行中与主对角线紧邻的右次对角线元素右边的元素变为零。

对于每一个变换 $\boldsymbol{V}_j$ 具有如下形式：

$$\boldsymbol{I}-\rho\boldsymbol{V}_j\boldsymbol{V}_j^{\mathrm{T}}$$

其中 ρ 为一个比例因子，以避免计算过程中的溢出现象与误差的积累。$\boldsymbol{V}_j$ 是一个列向量，即

$$\boldsymbol{V}_j = (v_0, v_1, \cdots, v_{n-1})^{\mathrm{T}}$$

则

$$\boldsymbol{AV}_j = \boldsymbol{A} - \rho \boldsymbol{V}_j \boldsymbol{V}_j^{\mathrm{T}} = \boldsymbol{A} - \boldsymbol{W}\boldsymbol{V}_j^{\mathrm{T}}$$

其中

$$\boldsymbol{W} = \rho \boldsymbol{AV}_j = \rho\left(\sum_{i=0}^{n-1} v_i a_{0i}, \sum_{i=0}^{n-1} v_i a_{1i}, \cdots, \sum_{i=0}^{n-1} v_i a_{m-1,i}\right)^{\mathrm{T}}$$

第二步　用变型的 *OR* 算法进行迭代，计算所有的奇异值。即用一系列的平面旋转变换将双对角线矩阵 $\boldsymbol{B}$ 逐步变成对角矩阵。

在每一次迭代中，用变换

$$\boldsymbol{B}' = \boldsymbol{U}_{p-1,p}^{\mathrm{T}} \cdots \boldsymbol{U}_{12}^{\mathrm{T}} \boldsymbol{U}_{01}^{\mathrm{T}} \boldsymbol{B} \boldsymbol{V}_{01} \boldsymbol{V}_{12} \cdots \boldsymbol{V}_{m-2,m-1}$$

其中变换 $\boldsymbol{U}_{j,j+1}^{\mathrm{T}}$ 将 $\boldsymbol{B}$ 中第 j 列主对角线以下的一个非零元素变为零，同时在第 j 行的次对角线元素的右边出现一个非零元素；而变换 $\boldsymbol{V}_{j,j+1}$ 将第 $j-1$ 行的次对角线元素右边的一个非零元素变为零，同时在第 j 列的主对角线元素的下方出现一个非零元素。由此可知，经过一次迭代 $(j=0,1,\cdots,p-1)$ 后，$\boldsymbol{B}'$ 仍为双对角线矩阵。但随着迭代的进行，最后收敛为对角矩阵，其对角线上的元素即为奇异值。

在每次迭代时，经过初始变换 $\boldsymbol{V}_{01}$ 后，将在第 0 列的主对角线下方出现一个非零元素。在变换 $\boldsymbol{V}_{01}$ 中，选择位移值 u 的计算公式如下：

$$b = [(s_{p-1} + s_p)(s_{p-1} - s_p) + e_{p-1}^2]/2$$

$$c = (s_p e_{p-1})^2$$

$$d = \mathrm{sign}(b)\sqrt{b^2 + c}$$

$$u = s_p^2 - c/(b + d)$$

最后还需要对奇异值按非递增次序进行排序。

在上述变换过程中，若对于某个次对角线元素 e_j 满足 $|e_j| \leqslant \varepsilon(|s_{j+1}| + |s_j|)$，则可认为 e_j 为零。

若对角线元素 s_j 满足 $|s_j| \leqslant \varepsilon(|e_{j-1}| + |e_j|)$，则可认为 s_j 为零（即为零奇异值）。其中 ε 为给定的精度要求。

3. 函数原型

```
//一般实矩阵的奇异值分解
template <class _Ty>
int MatrixSingularValue(matrix<_Ty>& a, matrix<_Ty>& u,
                                        matrix<_Ty>& v, _Ty eps);
```

4. 参数

a：matrix 浮点类型，体积为 $m \times n$。存放 $m \times n$ 的实矩阵 $\boldsymbol{A}$；返回时其对角线依次给出奇异值（以非递增次序排列），其余元素均为零。

u：matrix 浮点类型，体积为 $m \times m$。返回时存放左奇异向量 $\boldsymbol{U}$ 中。

v：matrix 浮点类型，体积为 $n \times n$。返回时存放右奇异向量 $\boldsymbol{V}^{\mathrm{T}}$ 中。

eps：浮点类型，为给定的精度要求。

5. 返回值

int 类型。

0：表示出现了迭代 60 次还未求得某个奇异值的情况，此时，矩阵的分解式为 $\boldsymbol{UAV}$。

>0：表示正常返回。

6. 函数说明

程序调用了两个辅助函数_MSV_1()和_MSV_2(),详见本节函数定义部分。

7. 示例

求下列两个矩阵 A 与 B 的奇异分解式 UAV 与 UBV(取 $\varepsilon = 0.000001$):

$$A = \begin{bmatrix} 1 & 1 & -1 \\ 2 & 1 & 0 \\ 1 & -1 & 0 \\ -1 & 2 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 1 & -1 & -1 \\ 2 & 1 & 0 & 2 \\ 1 & -1 & 0 & 1 \end{bmatrix}$$

示例程序如下:

```
//MatrixSingularValue.cpp 一般实矩阵的奇异值分解

#include <iostream>                    //输入输出流
#include "Matrix.h"                    //矩阵类及相关函数等的定义
using namespace std;                   //名字空间

void main()                            //定义控制台应用程序的入口点
{
    double a[4][3] = { {1.0,1.0,-1.0},{2.0,1.0,0.0},
                       {1.0,-1.0,0.0},{-1.0,2.0,1.0}};
    double b[3][4] = { {1.0,1.0,-1.0,-1.0},{2.0,1.0,0.0,2.0},
                       {1.0,-1.0,0.0,1.0}};

matrix<double> ma(&a[0][0], 4, 3);
matrix<double> mb(&b[0][0], 3, 4);
matrix<double> mu(4, 4), mv(3, 3), mc(4, 3), md(3, 4);

double eps(DOUBLEERROR);
int i = MatrixSingularValue(ma,mu,mv,eps);

cout << "EXAMPLE(1) " << endl;
cout << endl << "i = " << i << endl;
cout << endl << "MAT U IS: " << endl;
MatrixLinePrint(mu);

cout << endl << "MAT V IS: " << endl;
MatrixLinePrint(mv);

cout << endl << "MAT A IS: " << endl;
MatrixLinePrint(ma);

cout << endl << "MAT UAV IS: " << endl;
mc = mu * ma;
ma = mc * mv;
```

```
MatrixLinePrint(ma);

cout << endl << "EXAMPLE(2)" << endl << endl;
i = MatrixSingularValue(mb,mv,mu,eps);
cout << "i = " << i << endl << endl;

cout << "MAT U IS: " << endl;
MatrixLinePrint(mv);

cout << endl << "MAT V IS: " << endl;
MatrixLinePrint(mu);

cout << endl << "MAT B IS: " << endl;
MatrixLinePrint(mb);

cout << endl << "MAT UBV IS: " << endl;
md = mv * mb;
mb = md * mu;
MatrixLinePrint(mb);
cout << endl;
}
```

8. 示例结果

```
EXAMPLE(1)

i = 1

MAT U IS:
        0.276393        0.507093        0.723607        0
        0.447214        0.676123       -0.447214        0
        0.447214       -0.169031       -0.447214        0
       -0.723607        0.507093       -0.276393        0

MAT V IS:
        0.835549       -0.417775          -0.356822
        0.447214        0.894427       3.69049e-015
       -0.319151        0.159576          -0.934172

MAT A IS:
       2.80252               0               0
             0         2.64575               0
             0               0         1.07047
             0               0               0

MAT UAV IS:
```

```
        1         1                -1
        2         1     1.22125e-015
        1        -1     7.77156e-016
       -1         2                1

EXAMPLE(2)

i = 1

MAT U IS:
      -0.0884996       -0.90288       -0.420685
       -0.925355      -0.0817766        0.370177
       -0.368628        0.422043       -0.828247

MAT V IS:
       -0.719419       -0.201136       0.0275879       -0.664243
       -0.301829       -0.658891        0.422904         0.54398
       -0.473992         0.72485        0.392077        0.310161
       -0.408248               0       -0.816497        0.408248

MAT B IS:
        3.20792    5.38395e-020                0        0
              0         2.13495    -1.6979e-016        0
              0               0         1.07297        0

MAT UBV IS:
        1         1                -1        -1
        2         1     1.11022e-016         2
        1        -1     2.22045e-016         1
```

9. 示例说明

示例运行结果中有的元素应为 0 值，实际计算结果是一些很小的数，而非严格的 0 值，这是电子数字计算机进行数值计算本身性质所决定的。

5.6.5 广义逆的奇异值分解

1. 功能

利用奇异值分解一般 $m\times n$ 阶实矩阵 $\boldsymbol{A}$ 的广义逆 $\boldsymbol{A}^{+}$。

2. 算法

设 $m\times n$ 矩阵 $\boldsymbol{A}$ 的奇异值分解式为

$$\boldsymbol{A}=\boldsymbol{U}\begin{bmatrix}\sum & 0\\ 0 & 0\end{bmatrix}\boldsymbol{V}^{\mathrm{T}}$$

其中 $\sum=\mathrm{diag}(\sigma_0,\sigma_1,\cdots,\sigma_p)(p\leqslant\min(m,n)-1)$，且 $\sigma_0\geqslant\sigma_1\geqslant\cdots\geqslant\sigma_p>0$

关于奇异值分解参看本章 5.6.4 节的算法说明。

设 $\boldsymbol{U}=(\boldsymbol{U}_1,\boldsymbol{U}_2)$，其中 $\boldsymbol{U}_1$ 为 $\boldsymbol{U}$ 中前 $p+1$ 列正交向量组构成的 $m\times(p+1)$ 矩阵；$\boldsymbol{V}=(\boldsymbol{V}_1,\boldsymbol{V}_2)$，其中 $\boldsymbol{V}_1$ 为 $\boldsymbol{V}$ 中前 $p+1$ 列正交向量组构成的 $n\times(p+1)$ 矩阵，则 $\boldsymbol{A}$ 的广义逆为

$$\boldsymbol{A}^{+}=\boldsymbol{V}_1\sum{}^{-1}\boldsymbol{U}_1^{\mathrm{T}}$$

3. 函数原型

```
//广义逆的奇异值分解
template <class _Ty>
int GeneralizedInversionSingularValue(matrix<_Ty>& a, matrix<_Ty>& aa,
                                      _Ty eps, matrix<_Ty>& u, matrix<_Ty>& v);
```

4. 参数

a：matrix 浮点类型，体积为 $m\times n$。存放 $m\times n$ 的矩阵 $\boldsymbol{A}$；返回时其对角线依次给出奇异值，其余元素均为零。

aa：matrix 浮点类型，体积为 $n\times m$。返回时存放矩阵 $\boldsymbol{A}$ 的广义逆 $\boldsymbol{A}^{+}$。

eps：浮点类型变量。奇异值分解时的控制精度要求。

u：matrix 浮点类型，体积为 $m\times m$。返回时存放奇异值分解式中的左奇异向量 $\boldsymbol{U}$。

v：matrix 浮点类型，体积为 $n\times n$。返回时存放奇异值分解式中的右奇异向量 $\boldsymbol{V}^{\mathrm{T}}$。

5. 返回值

int 类型。

0：表示出现了迭代 60 次还未满足精度要求。

>0：表示正常返回。

6. 函数说明

本函数要调用一般实矩阵的奇异值分解的函数 MatrixSingularValue()(参看本章 5.6.4 节)。

7. 示例

求下列 5×4 阶矩阵 $\boldsymbol{A}$ 的广义逆 $\boldsymbol{A}^{+}$，再求 $\boldsymbol{A}^{+}$ 的广义逆 $(\boldsymbol{A}^{+})^{+}$：

$$\boldsymbol{A}=\begin{bmatrix}1 & 2 & 3 & 4\\ 6 & 7 & 8 & 9\\ 1 & 2 & 13 & 0\\ 16 & 17 & 8 & 9\\ 2 & 4 & 3 & 4\end{bmatrix}$$

取 $\varepsilon=0.000001$。

示例程序如下：

```
//GeneralizedInversionSingularValue.cpp 广义逆的奇异值分解

#include <iostream>                    //输入输出流
#include "Matrix.h"                    //矩阵类及相关函数等的定义
using namespace std;                   //名字空间

void main()                            // 定义控制台应用程序的入口点
{
    double a[5][4]={ {1.0,2.0,3.0,4.0},
                     {6.0,7.0,8.0,9.0},{1.0,2.0,13.0,0.0},
```

```
                {16.0,17.0,8.0,9.0},{2.0,4.0,3.0,4.0}};

matrix<double> ma(&a[0][0], 5, 4);
matrix<double> aa(4, 5), mc(5, 4), mu(5, 5), mv(4, 4);

double eps(DOUBLEERROR);
int m(5), n(4), ka(6);

cout << "MAT A IS: " << endl;
MatrixLinePrint(ma);

cout << endl << "MAT A+ IS: " << endl;
int i = GeneralizedInversionSingularValue(ma,aa,eps,mu,mv);
MatrixLinePrint(aa);

cout << endl << "MAT A+ + IS: " << endl;
i = GeneralizedInversionSingularValue(aa,mc,eps,mv,mu);
MatrixLinePrint(mc);
cout << endl;
}
```

8. 示例结果

```
MAT A IS:
        1         2         3         4
        6         7         8         9
        1         2        13         0
       16        17         8         9
        2         4         3         4

MAT A+ IS:
         -0.113487         0.264449        -0.0297724         0.0451837        -0.583187
          0.0718009       -0.347061         0.0286578         0.0490204         0.598791
        -0.00385819        0.0336949        0.0747631        -0.0112316        -0.0466841
            0.05798        0.160308        -0.0679684        -0.0534359        -0.0484417

MAT A+ + IS:
        1         2         3         4
        6         7         8         9
        1         2        13         0
       16        17         8         9
        2         4         3         4
```

第 6 章　矩阵特征值与特征向量的计算

本章介绍矩阵特征值与矩阵特征向量的计算算法，每种算法都用C++语言以函数的形式实现。6.1 节介绍矩阵特征值与矩阵特征向量的基本概念，以及计算矩阵特征值与矩阵特征向量的算法程序原型头文件，6.2 节中给出了计算矩阵特征值与矩阵特征向量的使用示例及相关说明。

6.1　矩阵特征值、特征向量度与头文件

本节介绍矩阵特征值与矩阵特征向量的基本概念，以及计算矩阵特征值与矩阵特征向量的算法程序原型，这些算法函数程序原型包含在头文件 random. h 中，函数的定义内容放在 random. inl 文件中。

本章介绍的计算矩阵特征值与矩阵特征向量算法及程序包括：

- 约化对称阵为对称三对角阵的豪斯荷尔德变换法；
- 实对称三角阵全部特征值及特征向量 *QR* 法；
- 约化一般实矩阵为赫申伯格阵的初等相似变换法；
- 求赫申伯格阵全部特征值 *QR* 法；
- 对称阵特征值及特征向量雅可比法；
- 实对称阵特征值及特征向量雅可比过关法。

6.1.1　基本概念

1. 矩阵特征值与矩阵特征向量

对 n 阶方阵

$$\boldsymbol{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}_{n\times n}$$

和 n 维非零列向量

$$\boldsymbol{\alpha}=(\alpha_1,\alpha_2,\cdots,\alpha_n)^{\mathrm{T}}$$

如果有一个数 λ，使得

$$\boldsymbol{A\alpha}=\lambda\boldsymbol{\alpha}$$

则称 λ 为矩阵 $\boldsymbol{A}$ 的特征值(特征根)，$\boldsymbol{\alpha}$ 为矩阵 $\boldsymbol{A}$ 的特征值 λ 所对应的特征向量。

矩阵 $\boldsymbol{A}$ 的所有特征值中绝对值最大的一个称为 $\boldsymbol{A}$ 的第一特征值。

2. 特征矩阵、特征多项式与特征方程

对 n 阶方阵

$$\boldsymbol{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}_{n\times n}$$

的特征矩阵定义为

$$\boldsymbol{A}-\lambda\boldsymbol{I}=\begin{bmatrix} a_{11}-\lambda & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22}-\lambda & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn}-\lambda \end{bmatrix}$$

式中 $\boldsymbol{I}$ 为 n 阶单位阵，行列式 $|\boldsymbol{A}-\lambda\boldsymbol{I}|$ 称为矩阵 $\boldsymbol{A}$ 的特征多项式，记作

$$\varphi(\lambda)=|\boldsymbol{A}-\lambda\boldsymbol{I}|$$

方程

$$\varphi(\lambda)=0$$

称为矩阵 $\boldsymbol{A}$ 的特征方程。

6.1.2 头文件

```
// EigenvalueVector.h    计算特征值特征向量头文件
// Ver 1.0.0.0
// 版权所有(C) 何渝，2002
// 最后修改：2002.8.10

#ifndef _EIGENVALUEVECTOR_H    //避免多次编译
#define _EIGENVALUEVECTOR_H

#include "Matrix.h"  //矩阵类及相关函数等的定义
#include <comm.h>  //公共头文件
#include <math.h>  //数学头文件

using namespace std;  //名字空间
//template <class _Ty = float>

//约化对称阵为对称三对角阵的豪斯荷尔德变换法
template <class _Ty>
int HouseholderTransform(matrix<_Ty>& a, matrix<_Ty>& q,
                         valarray<_Ty>& b, valarray<_Ty>& c);

//实对称三角阵全部特征值及特征向量 QR 法
template <class _Ty>
int EigenvalueVectorRealTriangleQR(valarray<_Ty>& b,
                    valarray<_Ty>& c, matrix<_Ty>& q, _Ty eps, int l);

//约化一般实矩阵为赫申伯格阵的初等相似变换法
```

```
//矩阵类型应是浮点型
template <class _Ty>
int HessenbergTransform(matrix<_Ty>& a);

//求赫申伯格阵全部特征值 QR 法
template <class _Ty>
int EigenvalueVectorHessenbergQR(matrix<_Ty>& a,
                    valarray<complex<_Ty> >& uv,_Ty eps, int jt);

//实对称阵特征值及特征向量雅可比法
template <class _Ty>
int EigenvalueVectorRealSymmetryJacobi(matrix<_Ty>& a,
                                       matrix<_Ty>& v, _Ty eps, int jt);

//实对称阵特征值及特征向量雅可比过关法
template <class _Ty>
int EigenvalueVectorRealSymmetryJacobiB(matrix<_Ty>& a,
                                        matrix<_Ty>& v, _Ty eps);

# include "EigenvalueVector. inl"   //类及相关函数的定义头文件

# endif    // _EIGENVALUEVECTOR_H
```

6.2 矩阵特征值与特征向量计算

6.2.1 约化对称阵为对称三对角阵的豪斯荷尔德变换法

1. 功能

用豪斯荷尔德(Householder)变换将 n 阶实对称矩阵约化为对称三对角阵。

2. 算法

约化 n 阶实对称矩阵 $\boldsymbol{A}$ 为对称三角阵的豪斯荷尔德方法，就是使 $\boldsymbol{A}$ 经过 $n-2$ 次正交变换后变成三对角阵 $\boldsymbol{A}_{n-2}$。即

$$\boldsymbol{A}_{n-2} = \boldsymbol{P}_{n-3}\cdots\boldsymbol{P}_1\boldsymbol{P}_0\boldsymbol{A}\boldsymbol{P}_0\boldsymbol{P}_1\cdots\boldsymbol{P}_{n-3}$$

每一次的正交变换 $\boldsymbol{P}_i(i=0,1,\cdots,n-3)$ 具有如下形式

$$\boldsymbol{P}_i = \boldsymbol{I} - \boldsymbol{U}_i\boldsymbol{U}_i^{\mathrm{T}}/\boldsymbol{H}_i$$

其中 $\boldsymbol{P}_i$ 为对称正交矩阵，且

$$\boldsymbol{H}_i = \frac{1}{2}\boldsymbol{U}_i^{\mathrm{T}}\boldsymbol{U}_i$$

$$\boldsymbol{U}_i = (a_{i0}^{(i)}, a_{i1}^{(i)}, \cdots, a_{i,l-2}^{(i)}, a_{i,l-1}^{(i)} \pm \sigma_i^{1/2}, 0, \cdots, 0)^{\mathrm{T}}$$

$$\sigma_i = (a_{i0}^{(i)})^2 + (a_{i1}^{(i)})^2 + \cdots + (a_{i,l-1}^{(i)})^2$$

式中 $l=n-i-1$。

对 $\boldsymbol{A}$ 的每一次变换为

$$\boldsymbol{A}_{i+1} = \boldsymbol{P}_i\boldsymbol{A}_i\boldsymbol{P}_i = (\boldsymbol{I} - \boldsymbol{U}_i\boldsymbol{U}_i^{\mathrm{T}}/\boldsymbol{H}_i)\boldsymbol{A}_i(\boldsymbol{I} - \boldsymbol{U}_i\boldsymbol{U}_i^{\mathrm{T}}/\boldsymbol{H}_i)$$

若令

$$\begin{cases} s_i = \boldsymbol{A}_i\boldsymbol{U}_i/\boldsymbol{H}_i \\ k_i = \boldsymbol{U}_i^{\mathrm{T}}s_i/2\boldsymbol{H}_i \\ q_i = s_i - k_i\boldsymbol{U}_i \end{cases}$$

则有

$$\boldsymbol{A}_{i+1} = \boldsymbol{A}_i - \boldsymbol{U}_i q_i^{\mathrm{T}} - q_i\boldsymbol{U}_i^{\mathrm{T}}$$

其中 s_i 的形式为

$$s_i = (s_{i0}, s_{i1}, \cdots, s_{il}, 0, \cdots, 0)^{\mathrm{T}}$$

q_i 的形式为

$$q_i = (s_{i0} - k_i u_{i0}, s_{i1} - k_i u_{i1}, \cdots, s_{i,i-1} - k_i u_{i,i-1}, s_{il}, 0, \cdots, 0)^{\mathrm{T}}$$

3. 函数原型

```
//约化对称阵为对称三对角阵的豪斯荷尔德变换法
template <class _Ty>
int HouseholderTransform(matrix<_Ty>& a, matrix<_Ty>& q,
                                    valarray<_Ty>& b, valarray<_Ty>& c);
```

4. 参数

a：matrix<_Ty>类型引用，体积为 $n\times n$。存放 n 阶实对称矩阵 $\boldsymbol{A}$。

q：matrix<_Ty>类型引用，体积为 $n\times n$。返回豪斯荷尔德变换的乘积体积为 $\boldsymbol{Q}$。在与本章 6.2.2 节中的函数 EigenvalueVectorRealTriangleQR()联用时，若将 $\boldsymbol{Q}$ 矩阵作为函数 EigenvalueVectorRealTriangleQR()中的一个参数，可计算实对称矩阵 $\boldsymbol{A}$ 的特征值及相应的特征向量。

b：valarray<_Ty>类型引用，长度为 n。返回对称三对角阵中的主对角线元素。

c：valarray<_Ty>类型引用，长度为 n。其中前 $n-1$ 个元素返回对称三对角阵中的次对角线元素。

由上可知本函数返回的对称三对角阵为

$$\boldsymbol{T} = \begin{bmatrix} b_0 & c_0 & & & & \\ c_0 & b_1 & c_1 & & 0 & \\ & c_1 & b_2 & c_2 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & c_{n-3} & b_{n-2} & c_{n-2} \\ & & & & c_{n-2} & b_{n-1} \end{bmatrix}$$

5. 返回值

int 类型。

1：正常返回，约化成功。

0：矩阵 a 不是对称阵。

6. 函数说明

本函数与 6.2.2 节的函数 EigenvalueVectorRealTriangleQR()联用，可以计算实对称矩阵 A 的全部特征值与相应的特征向量。

矩阵 a、q，数组 b、c 的数据类型应是浮点类型。

7. 示例

用豪斯荷尔德变换将下列 5 阶实对称矩阵约化为对称三对角阵，并给出豪斯荷尔德变换的乘积矩阵 Q：

$$A=\begin{bmatrix}10 & 1 & 2 & 3 & 4\\ 1 & 9 & -1 & 2 & -3\\ 2 & -1 & 7 & 3 & -5\\ 3 & 2 & 3 & 12 & -1\\ 4 & -3 & -5 & -1 & 15\end{bmatrix}$$

示例程序如下：

```
// HouseholderTransform.cpp  约化对称阵为对称三对角阵的豪斯荷尔德变换法

#include <iostream>              //输入输出流头文件
#include "Matrix.h"              //矩阵类及相关函数头文件
#include "EigenvalueVector.h"    //特征值及特征向量头文件
using namespace std;             //名字空间
void main()                      //定义控制台应用程序的入口点
{
  double a[5][5] =
  {
       {10.0, 1.0,   2.0,   3.0,   4.0},
       { 1.0, 9.0,   -1.0, 2.0,   -3.0},
       { 2.0, -1.0, 7.0,   3.0,   -5.0},
       { 3.0, 2.0,   3.0,   12.0,  -1.0},
       { 4.0, -3.0, -5.0, -1.0, 15.0}
  };

  valarray<double> b(5), c(5);
  matrix<double> q(5,5), da(&a[0][0],5,5);

  cout.setf(ios::fixed);  //输出数据为定点法
  cout.precision(6);      //精度 6 位

  if(HouseholderTransform(da,q,b,c) > 0)
  {
      cout << "MATRIX A IS: " << endl;
      MatrixLinePrint(da);
      cout << endl;

      cout << "MATRIX Q IS: " << endl;
      MatrixLinePrint(q);
      cout << endl;

      cout << "MATRIX B IS: " << endl;
```

```
        ValarrayPrint(b);
        cout << endl;

        cout << "MATRIX C IS: " << endl;
        ValarrayPrint(c);
        cout << endl;
    }
    else
        cout << "Matrix A is not symmetry." << endl;
}
```

8. 示例结果

```
MATRIX A IS:
    10.000000     1.000000     2.000000     3.000000     4.000000
     1.000000     9.000000    -1.000000     2.000000    -3.000000
     2.000000    -1.000000     7.000000     3.000000    -5.000000
     3.000000     2.000000     3.000000    12.000000    -1.000000
     4.000000    -3.000000    -5.000000    -1.000000    15.000000

MATRIX Q IS:
    -0.038454    -0.826960     0.030549     0.560112     0.000000
    -0.868565    -0.240190     0.106922    -0.420084     0.000000
     0.449244    -0.503710    -0.232936    -0.700140     0.000000
     0.205658    -0.068717     0.966113    -0.140028     0.000000
     0.000000     0.000000     0.000000     0.000000     1.000000

MATRIX B IS:
     9.295202    11.626712    10.960439     6.117647    15.000000

MATRIX C IS:
    -0.749485    -4.496268    -2.157041     7.141428     0.000000
```

即返回的三对角阵为：

$$T=\begin{bmatrix} 9.295202 & -0.7494847 & 0.000000 & 0.000000 & 0.000000 \\ -0.7494847 & 11.62671 & -4.496268 & 0.000000 & 0.000000 \\ 0.000000 & -4.496268 & 10.96044 & -2.157041 & 0.000000 \\ 0.000000 & 0.000000 & -2.157041 & 6.117647 & 7.141428 \\ 0.000000 & 0.000000 & 0.000000 & 7.141428 & 15.000000 \end{bmatrix}$$

6.2.2 实对称三角阵全部特征值及特征向量 *QR* 法

1. 功能

用变形 *QR* 方法计算实对称三对角阵的全部特征值与相应的特征向量。

2. 算法

有关 *QR* 方法的说明参看本章 6.2.4 节。

3. 函数原型

```
//实对称三角阵全部特征值及特征向量 QR 法
template <class _Ty>
int EigenvalueVectorRealTriangleQR(valarray<_Ty>& b,
                    valarray<_Ty>& c, matrix<_Ty>& q, _Ty eps, int l);
```

4. 参数

b:valarray<_Ty>类型引用,长度为 n。存放 n 阶对称三对角阵的主对角线上的各元素;返回时存放全部特征值。

c:valarray<_Ty>类型引用,长度为 n。前 $n-1$ 个元素存放 n 阶对称三对角阵的次对角线上的元素。

由参数 b 与 c 的说明可知,n 阶实对称三对角阵为

$$\boldsymbol{T}=\begin{bmatrix} b_0 & c_0 & & & & \\ c_0 & b_1 & c_1 & & 0 & \\ & c_1 & b_2 & c_2 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & c_{n-3} & b_{n-2} & c_{n-2} \\ & & & & c_{n-2} & b_{n-1} \end{bmatrix}$$

q:matrix<_Ty>类型引用,体积为 $n\times n$。若存放单位矩阵,则返回 n 阶实对称三对角阵 $\boldsymbol{T}$ 的特征向量组;若存放由本章 6.2.1 节中函数 HouseholderTransform()所返回的一般实对称矩阵 $\boldsymbol{A}$ 的豪斯荷尔德变换的乘积体矩阵 $\boldsymbol{Q}$,则返回实对称矩阵 $\boldsymbol{A}$ 的特征向量组。其中,q 中的第 j 列为与数组 b 中第 j 个特征值对应的特征向量。

eps:_Ty 类型变量。函数在迭代过程中的控制精度要求。

l:整型变量。为求得一个特征值所允许的最大迭代次数。

5. 返回值

int 类型。

<0:表示迭代了 l 次还未求得一个特征值,输出信息"fail"。

>0:表示正常返回。

6. 函数说明

本函数如果与本章 6.2.1 节的函数 HouseholderTransform()联用,则可以计算一般实对称矩阵的全部特征值与相应的特征向量。

矩阵 q,数组 b、c 及精度控制变量 eps 的数据类型应是浮点类型。

7. 示例

与本章 6.2.1 节中的函数 HouseholderTransform()联用,计算下列 5 阶实对称矩阵 $\boldsymbol{A}$ 的全部特征值与相应的特征向量:

$$\boldsymbol{A}=\begin{bmatrix} 10 & 1 & 2 & 3 & 4 \\ 1 & 9 & -1 & 2 & -3 \\ 2 & -1 & 7 & 3 & -5 \\ 3 & 2 & 3 & 12 & -1 \\ 4 & -3 & -5 & -1 & 15 \end{bmatrix}$$

取 $\varepsilon=0.000001$,最大迭代次数为 60。

示例程序如下：

```
//EigenvalueVectorRealTriangleQR.cpp  实对称三角阵全部特征值及特征向量 QR 法

#include <iostream>          //输入输出流头文件
#include "Matrix.h"          //矩阵类及相关函数头文件
#include "EigenvalueVector.h" //特征值及特征向量头文件
using namespace std;         //名字空间

void main()                  // 定义控制台应用程序的入口点
{
  double a[5][5] =
  {
       {10.0, 1.0,   2.0,   3.0,   4.0},
      { 1.0,  9.0,   -1.0, 2.0,   -3.0},
      { 2.0,  -1.0, 7.0,   3.0,   -5.0},
      { 3.0,  2.0,   3.0,   12.0,  -1.0},
       { 4.0, -3.0, -5.0, -1.0, 15.0}
  };

  valarray<double> b(5), c(5);
  matrix<double> q(5,5), da(&a[0][0],5,5);

  double eps = FLOATERROR;

  cout.setf(ios::fixed);  //输出数据为定点法
  cout.precision(6);      //精度 6 位

  HouseholderTransform(da,q,b,c);
  int k = EigenvalueVectorRealTriangleQR(b,c,q,eps,60);

  cout << "MATRIX A IS: " << endl;
  MatrixLinePrint(da);
  cout << endl;

  if(k>0)
  {
      cout << "MATRIX B IS: " << endl;
      ValarrayPrint(b);
      cout << endl;

      cout << "MATRIX Q IS: " << endl;
      MatrixLinePrint(q);
      cout << endl;
  }
```

```
}
```

8. 示例结果

```
MATRIX A IS:
    10.000000     1.000000     2.000000     3.000000     4.000000
     1.000000     9.000000    -1.000000     2.000000    -3.000000
     2.000000    -1.000000     7.000000     3.000000    -5.000000
     3.000000     2.000000     3.000000    12.000000    -1.000000
     4.000000    -3.000000    -5.000000    -1.000000    15.000000

MATRIX B IS:
    6.994838     9.365555     1.655266    15.808921    19.175420

MATRIX Q IS:
     0.654083     0.052151     0.387297    -0.623702     0.174505
     0.199681    -0.859964    -0.366221    -0.159101    -0.247303
     0.256510     0.505575    -0.704377    -0.227297    -0.361642
    -0.660403     0.000201     0.118926    -0.692684    -0.264411
    -0.174280    -0.046219    -0.453423    -0.232822     0.841244
```

6.2.3 约化一般实矩阵为赫申伯格阵的初等相似变换法

1. 功能

用初等相似变换将一般实矩阵约化为上 H 阵，即赫申伯格(Hessenberg)矩阵。

2. 算法

为了要将 n 阶实方阵 A 约化为上 H 阵，只需依次将 A 中的每一列约化为上 H 阵。为此，需要对于 $k=1,2,\cdots,n-2$ 作如下变换。

(1) 从第 $k-1$ 列的第 $k-1$ 个以下的元素中选取绝对值最大的元素 $a_{l,k-1}$。

(2) 交换第 l 行与 k 行；交换第 l 列与 k 列。

(3) 对于 $i=k+1,\cdots,n-1$ 作变换

$$
\begin{aligned}
&a_{l,k-1}/a_{k,k-1}\Rightarrow m, && 0\Rightarrow a_{i,k-1}\\
&a_{ij}-ma_{kj}\Rightarrow a_{ij}, && j=k,\cdots,n-1\\
&a_{jk}-ma_{ji}\Rightarrow a_{jk}, && j=0,1,\cdots,n-1
\end{aligned}
$$

3. 函数原型

```
//约化一般实矩阵为赫申伯格阵的初等相似变换法
template <class _Ty>
int HessenbergTransform(matrix<_Ty>& a);
```

4. 参数

a：matrix<_Ty>类型引用，体积为 $n\times n$。存放一般 n 阶实矩阵 A；返回时存放 A 的上 H 矩阵。

5. 返回值

int 类型。

0：要约化的实矩阵不是方阵。

1：程序正常结束，约化成功。

6. 函数说明

矩阵类型应是浮点型。

7. 示例

用初等相似变换将下列 5 阶矩阵 A 约化为上 H 阵：

$$A=\begin{bmatrix} 1 & 6 & -3 & -1 & 7 \\ 8 & -15 & 18 & 5 & 4 \\ -2 & 11 & 9 & 15 & 20 \\ -13 & 2 & 21 & 30 & -6 \\ 17 & 22 & -5 & 3 & 6 \end{bmatrix}$$

示例程序如下：

```
//HessenbergTransform.cpp  约化一般实矩阵为赫申伯格阵的初等相似变换法

#include <iostream>              //输入输出流头文件
#include "Matrix.h"              //矩阵类及相关函数头文件
#include "EigenvalueVector.h" //特征值及特征向量头文件
using namespace std;             //名字空间

void main()                      // 定义控制台应用程序的入口点
{
  double a[5][5] =
  {
      { 1.0, 6.0, -3.0, -1.0, 7.0},
      { 8.0, -15.0, 18.0, 5.0, 4.0},
      { -2.0, 11.0, 9.0, 15.0, 20.0},
      {-13.0, 2.0, 21.0, 30.0, -6.0},
      { 17.0, 22.0, -5.0, 3.0, 6.0}
  };

  matrix<double> da(&a[0][0],5,5);
  cout.setf(ios::fixed);  //输出数据为定点法
  cout.precision(6);      //精度 6 位
  int iRet = HessenbergTransform(da);

  if(iRet > 0)
  {
      cout << "MATRIX A IS: " << endl;
      MatrixLinePrint(da);
      cout << endl;
  }
}
```

8. 示例结果

```
      MATRIX A IS:
```

1.000000	10.941176	6.185671	8.267828	-3.000000
17.000000	14.647059	24.873047	25.779713	-5.000000
0.000000	-19.269896	35.009612	5.839105	17.176471
0.000000	0.000000	-61.373333	-45.554362	6.186748
0.000000	0.000000	0.000000	-22.852629	25.897692

6.2.4 求赫申伯格阵全部特征值 *QR* 法

1. 功能

用带原点位移的双重步 *QR* 方法计算实上 *H* 矩阵的全部特征值。

2. 算法

设上 *H* 矩阵 *A* 为不可约的，通过一系列双重步 *QR* 变换，使上 *H* 矩阵变为对角块全部是一阶块或二阶块，进而从中解出全部的特征值。

双重步 *QR* 变换的步骤可以归纳如下：

(1) 确定一个初等正交对称矩阵 Q_0，对 *A* 作相似变换

$$A_1 = Q_0 A Q_0$$

其中 Q_0 为对称正交矩阵，具有如下形式

$$Q_0 = \begin{bmatrix} \tilde{Q}_0 & 0 \\ 0 & I_{n-3} \end{bmatrix}$$

且 $\tilde{Q}_0$ 为 3×3 的矩阵。若令

$$\begin{cases} \alpha = a_{n-2,n-2} + a_{n-1,n-1} \\ \beta = a_{n-2,n-2} \cdot a_{n-1,n-1} - a_{n-2,n-1} \cdot a_{n-1,n-2} \end{cases}$$

$$\begin{cases} p_0 = a_{00}(a_{00} - \alpha) + a_{01} a_{10} + \beta \\ q_0 = a_{10}(a_{00} + a_{11} - \alpha) \\ r_0 = a_{10} a_{21} \end{cases}$$

且 $\tilde{Q}_0$ 中的各元素为

$$\tilde{Q}_0 = \begin{bmatrix} -\dfrac{p_0}{s_0} & -\dfrac{q_0}{s_0} & -\dfrac{r_0}{s_0} \\ -\dfrac{q_0}{s_0} & \dfrac{p_0}{s_0} + \dfrac{r_0^2}{s_0(p_0+s_0)} & -\dfrac{q_0 r_0}{s_0(p_0+s_0)} \\ -\dfrac{r_0}{s_0} & -\dfrac{q_0 r_0}{s_0(p_0+s_0)} & \dfrac{p_0}{s_0} + \dfrac{q_0^2}{s_0(p_0+s_0)} \end{bmatrix}$$

其中 $s_0 = \text{sign}(p_0)\sqrt{p_0^2 + q_0^2 + r_0^2}$。

由此可得

$$A_1 = Q_0 A Q_0 = \begin{bmatrix} * & * & * & * & \cdots & * \\ p_1 & * & * & * & \cdots & * \\ \underline{q_1} & * & * & * & \cdots & * \\ \underline{r_1} & \underline{*} & * & * & \cdots & * \\ & & & \ddots & \ddots & \vdots \\ & & 0 & & * & * \end{bmatrix}$$

其中有下划线的为次对角线以下新增加的三个非零元素。

(2) 利用同样的方法，依次确定正交对称矩阵 $\boldsymbol{Q}_1,\boldsymbol{Q}_2,\cdots,\boldsymbol{Q}_{n-2}$，并对 $\boldsymbol{A}_1,\boldsymbol{A}_2,\cdots,\boldsymbol{A}_{n-2}$ 作相似变换。

$$\boldsymbol{A}_{i+1}=\boldsymbol{Q}_i\boldsymbol{A}_i\boldsymbol{Q}_i,\quad i=1,2,\cdots,n-2$$

最后可得到上 $\boldsymbol{H}$ 矩阵

$$\boldsymbol{A}_{n-1}=\boldsymbol{Q}_{n-2}\boldsymbol{A}_{n-2}\boldsymbol{Q}_{n-2}$$

在这个过程中，一般有

$$\boldsymbol{A}_i=\begin{bmatrix}
* & * & * & \cdots & * & * & * & * & \cdots & * \\
* & * & * & \cdots & * & * & * & * & \cdots & * \\
 & * & * & \cdots & * & * & * & * & \cdots & * \\
 & & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots & & \vdots \\
 & & & * & * & * & * & * & \cdots & * \\
 & & & & p_i & * & * & * & \cdots & * \\
 & & & & \underline{q_i} & * & * & * & \cdots & * \\
 & & & & \underline{r_i} & \underline{*} & * & * & \cdots & * \\
 & & & & & & & \ddots & \ddots & \vdots \\
 & & & & & & & & * & *
\end{bmatrix}$$

为了消去 $\boldsymbol{A}_i$ 中次对角线以下有下划线的三个非零元素，$\boldsymbol{Q}_i$ 可以取如下形式的正交对称矩阵：

$$\boldsymbol{Q}_i=\begin{bmatrix}\boldsymbol{I}_i & 0 & 0\\ 0 & \tilde{\boldsymbol{Q}}_i^{(0)} & 0\\ 0 & 0 & \boldsymbol{I}_{n-i-3}\end{bmatrix}$$

其中 $\tilde{\boldsymbol{Q}}_i^{(0)}$ 为 3×3 的矩阵，且

$$\tilde{\boldsymbol{Q}}_i^{(0)}=\begin{bmatrix}
-\dfrac{p_i}{s_i} & -\dfrac{q_i}{s_i} & -\dfrac{r_i}{s_i}\\
-\dfrac{q_i}{s_i} & \dfrac{p_i}{s_i}+\dfrac{r_i^2}{s_i(p_i+s_i)} & -\dfrac{q_ir_i}{s_i(p_i+s_i)}\\
-\dfrac{r_i}{s_i} & -\dfrac{q_ir_i}{s_i(p_i+s_i)} & \dfrac{p_i}{s_i}+\dfrac{q_i^2}{s_i(p_i+s_i)}
\end{bmatrix}$$

其中 $s_i=\operatorname{sign}(p_i)\sqrt{p_i^2+q_i^2+r_i^2}$。

上式中的 $p_i,q_i,r_i(i=1,2,\cdots,n-2)$ 是从相应的 $\boldsymbol{A}_i$ 的第 $i-1$ 列中取得。即

$$p_i=a_{i,i-1}^{(i)},\quad q_i=a_{i+1,i-1}^{(i)},\quad r_i=a_{i+2,i-1}^{(i)}$$

而

$$p_{n-2}=a_{n-2,n-3}^{(n-2)},\quad q_{n-2}=a_{n-1,n-3}^{(n-2)},\quad r_{n-2}=0$$

反复进行以上两步，直到将上 $\boldsymbol{H}$ 矩阵 $\boldsymbol{A}$ 变为对角块全部是一阶块或二阶块为止，此时，就可以直接从各一阶块或二阶块中解出全部特征值。

QR 方法是一种迭代方法。在每次迭代计算过程中，如果次对角线元素的模小到一定程度，就可以把它们当作零看待，将矩阵分割为各不可约的上 $\boldsymbol{H}$ 矩阵，以便逐步降低主子阵的阶数，从而减少计算工作量。

在本函数中，判断对角线元素的模是否可认为是零的准则为

$$|a_{k,k-1}|\leqslant\varepsilon(|a_{k-1,k-1}|+|a_{kk}|)$$

当上 H 矩阵 A 类似正交阵时，本函数程序工作可能失败或误差较大，其原因是舍入误差的影响，但这种情况在一般工程应用中是少见的。

3. 函数原型

```
//求赫申伯格阵全部特征值 QR 法
template <class _Ty>
int EigenvalueVectorHessenbergQR(matrix<_Ty>& a,
                    valarray<complex<_Ty> >& uv, _Ty eps, int jt);
```

4. 参数

a：matrix<_Ty>类型引用，体积为 $n\times n$。存放上 H 矩阵 A。

uv：valarray<complex<_Ty> >类型引用，长度为 n。返回 n 个特征值。

eps：_Ty 类型变量。控制精度要求。

jt：整型变量。控制最大迭代次数。

5. 返回值

int 类型。

<0：表示迭代超过 jt 次而还未满足精度要求，输出信息“fail”。

>1：程序正常返回。

6. 函数说明

本函数如果与本章 6.2.3 节的函数 HessenbergTransform()联用，可以计算一般实对称矩阵的全部特征值。

矩阵 a，精度控制变量 eps 的数据类型应是浮点类型，数组 uv 的数据类型应为复数浮点类型。

7. 示例

与本章 6.2.3 节中的函数 HessenbergTransform()联用，计算下列 5 阶矩阵 A 的全部特征值：

$$A=\begin{bmatrix} 1 & 6 & -3 & -1 & 7 \\ 8 & -15 & 18 & 5 & 4 \\ -2 & 11 & 9 & 15 & 20 \\ -13 & 2 & 21 & 30 & -6 \\ 17 & 22 & -5 & 3 & 6 \end{bmatrix}$$

取 $\varepsilon=0.000001$，最大迭代次数为 60。

示例程序如下：

```
//EigenvalueVectorHessenbergQR.cpp  求赫申伯格阵全部特征值 QR 法

#include <iostream>          //输入输出流头文件
#include "Matrix.h"          //矩阵类及相关函数头文件
#include "EigenvalueVector.h" //特征值及特征向量头文件
using namespace std;         //名字空间

void main()                  // 定义控制台应用程序的入口点
{
  double a[5][5] =
```

```
    {
        { 1.0, 6.0, -3.0, -1.0, 7.0},
        { 8.0, -15.0, 18.0, 5.0, 4.0},
        { -2.0, 11.0, 9.0, 15.0, 20.0},
        { -13.0, 2.0, 21.0, 30.0, -6.0},
        { 17.0, 22.0, -5.0, 3.0, 6.0}
    };

    valarray<complex<double> > uv(5);
    matrix<double> da(&a[0][0],5,5);
    double eps = FLOATERROR;

    if(HessenbergTransform(da) > 0)
    {
        cout << "MATRIX H IS: " << endl;
        MatrixLinePrint(da);
        cout << endl;

        int k = EigenvalueVectorHessenbergQR(da,uv,eps,60);

        if(k>0)
        {
            cout << "EIGENVALUE IS: " << endl;
            ValarrayPrint(uv);
            cout << endl;
        }
    }
}
```

8. 示例结果

```
MATRIX H IS:
     1.000000    10.941176     6.185671     8.267828    -3.000000
    17.000000    14.647059    24.873047    25.779713    -5.000000
     0.000000   -19.269896    35.009612     5.839105    17.176471
     0.000000     0.000000   -61.373333   -45.554362     6.186748
     0.000000     0.000000     0.000000   -22.852629    25.897692

EIGENVALUE IS:
(-0.661716,0.000000) (42.961008,0.000000) (-15.339637,-6.755658) (-15.339637,6.755658)
(19.379983,0.000000)
```

6.2.5 实对称阵特征值及特征向量雅可比法

1. 功能

用雅可比(Jacobi)方法求实对称矩阵的全部特征值与相应的特征向量。

2. 算法

雅可比方法的基本思想如下：

设 n 阶矩阵 $\boldsymbol{A}$ 为对称矩阵。在 n 阶对称矩阵 $\boldsymbol{A}$ 的非对角线元素中选取一个绝对值最大的元素，设为 a_{pq}。利用平面旋转变换矩阵 $\boldsymbol{R}_0(p,q,\theta)$ 对 $\boldsymbol{A}$ 进行正交相似变换：

$$\boldsymbol{A}_1 = \boldsymbol{R}_0(p,q,\theta)^{\mathrm{T}}\boldsymbol{A}\boldsymbol{R}_0(p,q,\theta)$$

其中 $\boldsymbol{R}_0(p,q,\theta)$ 的元素为

$$r_{pp} = \cos\theta,\ r_{qq} = \cos\theta,\ r_{pq} = -\sin\theta,\ r_{qp} = \sin\theta$$
$$r_{ij} = 0, i,j \neq p,q$$

如果按下式确定 θ

$$\mathrm{tg}2\theta = \frac{2a_{pq}}{a_{pp} - a_{qq}}$$

则对称矩阵 $\boldsymbol{A}$ 经上述变换后，其非对角线元素的平方将减少 $2a_{pq}^2$，对角线元素的平方和增加 $2a_{pq}^2$，而矩阵中所有元素的平方和保持不变。由此可知，对称矩阵 $\boldsymbol{A}$ 每经过一次变换，其非对角线元素的平方和“向零接近了一步”。因此，只要反复进行上述变换，就可逐步将矩阵 $\boldsymbol{A}$ 变为对角矩阵。对角矩阵中对角线上的元素 $\lambda_0,\lambda_1,\cdots,\lambda_{n-1}$ 即为特征值，而每一步中的平面旋转矩阵的乘积的第 i 列($i=0,1,\cdots,n-1$)即为与 λ_i 对应的特征向量。

综上所述，用雅可比方法求 n 阶对称矩阵 $\boldsymbol{A}$ 的特征值及相应特征向量的步骤如下。

(1) 令 $\boldsymbol{S}=\boldsymbol{I}_n$($\boldsymbol{I}_n$ 为单位矩阵)。

(2) 在 $\boldsymbol{A}$ 中选取非对角线元素中绝对值最大者，设为 a_{pq}。

(3) 若 $|a_{pq}|<\varepsilon$，则迭代过程结束。此时对角线元素 $a_{pq}(i=0,1,\cdots,n-1)$ 即为特征值 λ_i，矩阵 $\boldsymbol{S}$ 的第 i 列为与 λ_i 对应的特征向量。否则，继续下一步。

(4) 计算平面旋转矩阵的元素及其变换后的矩阵 $\boldsymbol{A}_1$ 的元素，其计算公式如下：

$$x = -a_{pq}$$
$$y = \frac{1}{2}(a_{qq} - a_{pp})$$
$$\omega = \mathrm{sign}(y)\frac{x}{\sqrt{x^2+y^2}}$$
$$\sin 2\theta = \omega$$
$$\sin\theta = \frac{\omega}{\sqrt{2(1+\sqrt{1-\omega^2})}}$$
$$\cos\theta = \sqrt{1-\sin^2\theta}$$
$$a_{pp}^{(1)} = a_{pp}\cos^2\theta + a_{qq}\sin^2\theta + a_{pq}\sin 2\theta$$
$$a_{qq}^{(1)} = a_{pp}\sin^2\theta - a_{qq}\cos^2\theta - a_{pq}\sin 2\theta$$
$$a_{pq}^{(1)} = a_{qp}^{(1)} = 0$$
$$\left.\begin{aligned} a_{pj}^{(1)} &= a_{pj}\cos\theta + a_{qj}\sin\theta \\ a_{qj}^{(1)} &= -a_{pj}\sin\theta + a_{qj}\cos\theta \end{aligned}\right\} j \neq p,q$$
$$\left.\begin{aligned} a_{ip}^{(1)} &= a_{ip}\cos\theta + a_{iq}\sin\theta \\ a_{iq}^{(1)} &= -a_{iq}\sin\theta + a_{iq}\cos\theta \end{aligned}\right\} i \neq p,q$$
$$a_{ij}^{(1)} = a_{ij}, i,j \neq p,q$$

(5) $S = S \cdot R(p,q,\theta)$ 转(2)。

3. 函数原型

```
//实对称阵特征值及特征向量雅可比法
template <class _Ty>
int EigenvalueVectorRealSymmetryJacobi(matrix<_Ty>& a,
                                       matrix<_Ty>& v, _Ty eps, int jt)
```

4. 参数

a:matrix<_Ty>类型引用,体积为 $n\times n$。存放 n 阶实对称矩阵 $\boldsymbol{A}$;返回时,对角线上存放 n 个特征值。

v:matrix<_Ty>类型引用,体积为 $n\times n$。返回特征向量。其中第 i 列为与 λ_i(即返回的 a_{ii},$i=0,1,\cdots,n-1$)对应的特征向量。

eps:_Ty 类型变量。控制精度要求。

jt:整型变量。控制最大迭代次数。

5. 返回值

int 类型。

<0:表示迭代超过 jt 次而还未满足精度要求,输出信息"fail"。

0:矩阵 a 不是对称阵。

>0:程序正常返回。

本函数返回一个整型标志值。若返回的标志值小于 0,则表示迭代了 jt 次还达不到精度要求;若返回的标志值大于 0,则表示正常返回。

6. 函数说明

矩阵 a、v,精度控制变量 eps 的数据类型应是浮点类型。

7. 示例

用雅可比方法计算下列 3 阶实对称矩阵的全部特征值与相应的特征向量:

$$\boldsymbol{A}=\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

取 $\varepsilon=0.000001$,最大迭代次数为 100。

示例程序如下:

```
//EigenvalueVectorRealSymmetryJacobi.cpp  实对称阵特征值及特征向量雅可比法

#include <iostream>                    //输入输出流头文件
#include "Matrix.h"                    //矩阵类及相关函数头文件
#include "EigenvalueVector.h"          //特征值及特征向量头文件
using namespace std;                   //名字空间

void main()                            // 定义控制台应用程序的入口点
{
  double a[3][3] =
  {
      { 2.0, -1.0, 0.0},
      {-1.0, 2.0, -1.0},
```

```
        { 0.0, -1.0, 2.0}
    };

    matrix<double> v(3,3), da(&a[0][0],3,3);
    double eps = FLOATERROR;
    cout.setf(ios::fixed);  //输出数据为定点法
    cout.precision(6);      //精度 6 位

    int k = EigenvalueVectorRealSymmetryJacobi(da, v, eps, 100);
    if(k > 0)
    {
        cout << "MATRIX A IS: " << endl;
        MatrixLinePrint(da);
        cout << endl;
        cout << "MATRIX V IS: " << endl;
        MatrixLinePrint(v);
        cout << endl;
    }
}
```

8. 示例结果

```
MATRIX A IS:
    3.414214    0.000000    0.000000
    0.000000    0.585786    0.000000
    0.000000    0.000000    2.000000

MATRIX V IS:
    0.500000    0.500000   -0.707107
   -0.707107    0.707107   -0.000000
    0.500000    0.500000    0.707107
```

本问题的准确结果为：

特征值　　$\lambda_0 = 2+\sqrt{2}, \lambda_1 = 2-\sqrt{2}, \lambda_2 = 2.0$

特征向量

$$V_0 = \begin{bmatrix} 0.5 \\ -\sqrt{2}/2 \\ 0.5 \end{bmatrix}, \quad V_1 = \begin{bmatrix} 0.5 \\ \sqrt{2}/2 \\ 0.5 \end{bmatrix}, \quad V_2 = \begin{bmatrix} -\sqrt{2}/2 \\ 0 \\ \sqrt{2}/2 \end{bmatrix}$$

6.2.6　实对称阵特征值及特征向量雅可比过关法

1. 功能

用雅可比(Jacobi)过关法求实对称矩阵的全部特征值与相应的特征向量。

2. 算法

基本方法同本章 6.2.5 节。在选取非对角线上的绝对值最大的元素时改用如下方法：

首先计算实对称矩阵 A 的非对角线元素的平方和的平方根。

$$v_0 = \left(2\sum_{i=1}^{n-1}\sum_{j=0}^{i-1}a_{ij}^2\right)^{\frac{1}{2}}$$

然后设置关口 $v_1 = v_0/n$，在非对角线元素中按行扫描选取第一个绝对值大于或等于 v_1 的元素 a_{pq}，进行平面旋转变换，直到所有非对角线元素的绝对值均小于 v_1 为止，再设关口 $v_2 = v_1/n$，重复这个过程。以此类推，这个过程一直作到对于某个 $v_k < \varepsilon$ 为止。

3. 函数原型

```
//实对称阵特征值及特征向量雅可比过关法
template <class _Ty>
int EigenvalueVectorRealSymmetryJacobiB(matrix<_Ty>& a,
                                        matrix<_Ty>& v, _Ty eps)
```

4. 参数

a:matrix<_Ty>类型引用，体积为 $n \times n$。存放 n 阶实对称矩阵 $\boldsymbol{A}$；返回时，对角线上存放 n 个特征值。

v:matrix<_Ty>类型引用，体积为 $n \times n$。返回特征向量。其中第 i 列为与 λ_i（即返回的 a_{ii}，$i=0,1,\cdots,n-1$）对应的特征向量。

eps:_Ty 类型变量。控制精度要求。

5. 返回值

int 类型。

<0:表示矩阵 $\boldsymbol{A}$ 不是对称阵。

>0:程序正常返回。

6. 函数说明

矩阵 a、v，精度控制变量 eps 的数据类型应是浮点类型。

7. 示例

用雅可比过关法求下列 5 阶实对称矩阵 $\boldsymbol{A}$ 的全部特征值与相应的特征向量：

$$\boldsymbol{A} = \begin{bmatrix} 10 & 1 & 2 & 3 & 4 \\ 1 & 9 & -1 & 2 & -3 \\ 2 & -1 & 7 & 3 & -5 \\ 3 & 2 & 3 & 12 & -1 \\ 4 & -3 & -5 & -1 & 15 \end{bmatrix}$$

取 $\varepsilon = 0.000001$。

示例程序如下：

```
//EigenvalueVectorRealSymmetryJacobiB.cpp  实对称阵特征值及特征向量雅可比过关法

#include <iostream>              //输入输出流头文件
#include "Matrix.h"              //矩阵类及相关函数头文件
#include "EigenvalueVector.h"    //特征值及特征向量头文件
using namespace std;             //名字空间

void main()                      // 定义控制台应用程序的入口点
{
  double a[5][5] =
```

```
    {
        {10.0, 1.0, 2.0, 3.0, 4.0},
        { 1.0, 9.0, -1.0, 2.0, -3.0},
        { 2.0, -1.0, 7.0, 3.0, -5.0},
        { 3.0, 2.0, 3.0, 12.0, -1.0},
        { 4.0, -3.0, -5.0, -1.0, 15.0}
    };

    matrix<double> v(5,5), da(&a[0][0],5,5);
    double eps = FLOATERROR;

    cout.setf(ios::fixed);  //输出数据为定点法
    cout.precision(6);      //精度 6 位

    if(EigenvalueVectorRealSymmetryJacobiB(da,v,eps)>0)
    {
        cout << "MATRIX A IS: " << endl;
        MatrixLinePrint(da);
        cout << endl;

        cout << "MATRIX V IS: " << endl;
        MatrixLinePrint(v);
        cout << endl;
    }
}
```

8. 示例结果

```
MATRIX A IS:
      6.994838    0.000000    0.000000   -0.000000    0.000000
      0.000000    9.365555    0.000000    0.000000   -0.000000
      0.000000    0.000000    1.655266    0.000000   -0.000000
     -0.000000    0.000000    0.000000   15.808921    0.000000
      0.000000   -0.000000   -0.000000    0.000000   19.175420

MATRIX V IS:
      0.654083   -0.052151   -0.387297    0.623702    0.174505
      0.199681    0.859964    0.366221    0.159101   -0.247303
      0.256510   -0.505575    0.704377    0.227297   -0.361642
     -0.660403   -0.000201   -0.118926    0.692684   -0.264411
     -0.174280    0.046219    0.453423    0.232822    0.841244
```

第7章　线性方程组求解

本章介绍线性方程组求解的各种算法，每种算法都用C++语言以函数的形式实现。7.1节介绍线性方程组的基本概念，以及求解线性方程组各种算法程序原型的头文件，7.2节给出了求解线性方程组各种算法程序的使用示例及相关说明。

7.1　线性方程组与头文件

本节介绍线性方程组的基本概念和相关头文件，其各种算法的函数程序原型包含在头文件 Interpolation. h 中，函数的定义内容在 Interpolation. inl 文件中。

本章介绍的求解线性方程组的算法及程序包括：

- 全选主元高斯消元法；
- 全选主元高斯—约当消元法；
- 三对角方程组的追赶法；
- 一般带型方程组求解；
- 对称方程组的分解法；
- 对称正定方程组的平方根法；
- 大型稀疏方程组全选主元高斯—约当(Gauss-Jordan)法；
- 托伯利兹(Toeplitz)方程组的列文逊(Levinson)法；
- 高斯—赛德尔迭代法；
- 对称正定方程组的共轭梯度法；
- 线性最小二乘问题的豪斯荷尔德变换法；
- 线性最小二乘问题的广义逆法；
- 病态方程组求解。

7.1.1　基本概念

1. 一般情形的线性方程组

含 n 个未知量 m 个方程的线性方程组取如下形式：

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases} \tag{1}$$

记

$$\boldsymbol{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

$$x=(x_1,x_2,\cdots,x_n)^{\mathrm{T}}$$

$$b=(b_1,b_2,\cdots,b_m)^{\mathrm{T}}$$

式中的 T 表示转置，则给定线性方程组的矩阵形式为

$$\boldsymbol{A}x=b \tag{2}$$

相应的齐次方程组为

$$\boldsymbol{A}x=0 \tag{3}$$

$\boldsymbol{A}$ 称为方程组(1)的系数矩阵，

$$\boldsymbol{C}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{bmatrix}$$

称为方程组(1)的增广矩阵。

2. 含 n 个未知量 n 个方程的线性方程组

含 n 个未知量 n 个方程的线性方程组取如下形式：

$$\begin{cases} a_{11}x_1+a_{12}x_2+\cdots+a_{1n}x_n=b_1 \\ a_{21}x_1+a_{22}x_2+\cdots+a_{2n}x_n=b_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_{n1}x_1+a_{n2}x_2+\cdots+a_{nn}x_n=b_n \end{cases} \tag{4}$$

记

$$\boldsymbol{A}=(a_{ij})=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \ddots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

为方程组(4)的系数矩阵，记

$$x=(x_1,x_2,\cdots,x_n)^{\mathrm{T}}$$

为求知量向量，记

$$b=(b_1,b_2,\cdots,b_n)^{\mathrm{T}}$$

为常数项向量，式中的 T 表示转置，则线性方程组(4)可矩阵形式

$$\boldsymbol{A}x=b$$

7.1.2 Interpolation.h

```
//LinearEquation.h          线性方程(组)求解函数(方法)声明
// Ver 1.0.0.0
// 版权所有(C) 何渝，2002
// 最后修改：2002.5.31
```

```
#ifndef _LINEAREQUATION_H
#define _LINEAREQUATION_H

#include <valarray>          //模板类 valarray 的标准头文件
#include <Matrix.h>          //矩阵类头文件
#include <comm.h>            //公共头文件

//全选主元高斯消元法
template <class _Ty>
int LE_TotalChoiceGauss(matrix<_Ty>& a, valarray<_Ty>& b);

//全选主元高斯—约当消元法
template <class _Ty>
int LE_TotalChoiceGaussJordan(matrix<_Ty>& a, matrix<_Ty>& b);

//三对角方程组的追赶法
template <class _Ty>
int LE_TridiagonalEquationGauss(valarray<_Ty>& b, valarray<_Ty>& d);

//一般带型方程组求解
template <class _Ty>
int LE_StrapEquationGauss(matrix<_Ty>& b, matrix<_Ty>& d, int l, int il);

//对称方程组的分解法
template <class _Ty>
int LE_SymmetryEquation(matrix<_Ty>& a, matrix<_Ty>& c);

//对称正定方程组的平方根法
template <class _Ty>
int LE_SymmetryRegularEuationSquareRoot(matrix<_Ty>& a, matrix<_Ty>& d);

//大型稀疏方程组全选主元高斯—约当(Gauss-Jordan)法
template <class _Ty>
int LE_SparseEuationTotalChoiceGaussJordan(matrix<_Ty>& a, valarray<_Ty>& b);

//托伯利兹(Toeplitz)方程组的列文逊(Levinson)法
template <class _Ty>
int LE_ToeplitzEuationLevinson(valarray<_Ty>& t, valarray<_Ty>& b, valarray<_Ty>& x);

//高斯—赛德尔迭代法
template <class _Ty>
int LE_GaussSeidelIteration(matrix<_Ty>& a, valarray<_Ty>& b, valarray<_Ty>& x, _Ty eps);

//对称正定方程组的共轭梯度法
```

```
    template <class _Ty>
    int LE_SymmetryRegularEuationConjugateGradient(matrix<_Ty>& a, valarray<_Ty>& b, _Ty eps,
valarray<_Ty>& x);

    //线性最小二乘问题的豪斯荷尔德变换法
    template <class _Ty>
    int LE_LinearLeastSquareHouseholder(matrix<_Ty>& a, valarray<_Ty>& b, matrix<_Ty>& q);

    //线性最小二乘问题的广义逆法
    template <class _Ty>
    int LE_LinearLeastSquareGeneralizedInverse(matrix<_Ty>& a, valarray<_Ty>& b, matrix<_Ty>&
x, matrix<_Ty>& aa, _Ty eps, matrix<_Ty>& u, matrix<_Ty>& v);

    //病态方程组求解
    template <class _Ty>
    int LE_IllConditionedEquation(matrix<_Ty>& a, valarray<_Ty>& b, _Ty eps, matrix<_Ty>& x);

    #include <LinearEquation.inl>

    #endif //_LINEAREQUATION_H
```

7.2 求解线性方程组

7.2.1 全选主元高斯消元法

1. 功能

用全选主元高斯(Gauss)消元法求解 n 阶线性方程组 $\boldsymbol{AX}=\boldsymbol{B}$。其中

$$\boldsymbol{A}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix},\boldsymbol{X}=\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix},\boldsymbol{B}=\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

2. 算法

为保证数值计算的稳定性,本算法采用全选主元。

全选主元高斯消元法分两步进行。

第一步　对于 k 从 0 到 $n-2$ 进行以下三步

(1) 从系数矩阵 $\boldsymbol{A}$ 的第 k 行、第 k 列开始的右下角子阵中选取绝对值最大的元素,并通过行交换与列交换将它交换到主元素的位置上。

(2) 归一化

$$a_{kj}/a_{kk}\Rightarrow a_{kj},j=k+1,\cdots,n-1$$

$$b_k/b_{kk}\Rightarrow b_k$$

(3) 消元

$$a_{ij}-a_{ik}a_{kj}\Rightarrow a_{ij},i,j=k+1,\cdots,n-1$$

$b_i - a_{ik}b_k \Rightarrow b_i, i = k+1, \cdots, n-1$

第二步　回代过程

(1) $b_{n-1}/a_{n-1,n-1} \Rightarrow x_{n-1}$

(2) $b_i - \sum_{j=i+1}^{n-1} a_{ij}x_j \Rightarrow x_i, i = n-2, \cdots, 1, 0$

最后对解向量中的元素顺序进行调整。

3. 函数原型

```
//全选主元高斯消元法
template <class _Ty>
int LE_TotalChoiceGauss(matrix<_Ty>& a, valarray<_Ty>& b);
```

4. 参数

a:matrix<_Ty>类型引用,体积为 $n \times n$。存放方程组的系数矩阵,返回时将被破坏。

b:valarray<_Ty>类型引用,长度为 n。存放方程组右端的常数向量;返回方程组的解向量。

5. 返回值

int 类型。

0:矩阵为奇异阵。

1:程序正常结束。

6. 函数说明

LE_TotalChoiceGauss()全选主元高斯(Gauss)消元法的系数矩阵和右边向量可以是实数,也可以是复数。若是复数,解也是复数。

7. 示例

求解下列 4 阶实系数实右边方程组

$$\begin{cases} 0.2368x_0 + 0.2471x_1 + 0.2568x_2 + 1.2671x_3 = 1.8471 \\ 0.1968x_0 + 0.2071x_1 + 1.2168x_2 + 0.2271x_3 = 1.7471 \\ 0.1581x_0 + 1.1675x_1 + 0.1768x_2 + 0.1871x_3 = 1.6471 \\ 1.1161x_0 + 0.1254x_1 + 0.1397x_2 + 0.1490x_3 = 1.5471 \end{cases}$$

与复系数复右边方程组

$$\begin{cases} (1.0,3.0)x_0 + (3.0,-2.0)x_1 + (2.0,1.0)x_2 + (13.0,6.0)x_3 = (2.0,1.0) \\ (7.0,-2.0)x_0 + (2.0,7.0)x_1 + (1.0,5.0)x_2 + (-2.0,8.0)x_3 = (7.0,2.0) \\ (9.0,9.0)x_0 + (15.0,-3.0)x_1 + (3.0,15.0)x_2 + (-2.0,1.0)x_3 = (3.0,-2.0) \\ (-2.0,-2.0)x_0 + (-2.0,-2.0)x_1 + (11.0,7.0)x_2 + (5.0,6.0)x_3 = (9.0,3.0) \end{cases}$$

示例程序如下:

```
//LE_TotalChoiceGauss.cpp          全选主元高斯消元法

#include <iostream>                //输入输出流头文件
#include "LinearEquation.h"        //线性方程(组)求解头文件

void main()
{
  int i;
```

```
double a[4][4] =                    //实系数矩阵
  {
        {0.2368, 0.2471, 0.2568, 1.2671},
        {0.1968, 0.2071, 1.2168, 0.2271},
        {0.1581, 1.1675, 0.1768, 0.1871},
        {1.1161, 0.1254, 0.1397, 0.1490}
};

double b[4] = {1.8471, 1.7471, 1.6471, 1.5471};

matrix<double> aa(&a[0][0], 4, 4);
valarray<double> bb(b,4);

cout.setf(ios::fixed);              //输出数据为定点法
cout.precision(6);                  //精度6位

if(LE_TotalChoiceGauss(aa, bb) != 0)
{
  for(i = 0; i < 4; i++)            //输出实数解
  {
      cout << "x(" << i << ") = ";
      ValarrayPrint(bb, i);
  }
}
cout<<endl;

complex<double> ca[4][4]=      //复系数矩阵
{
    {complex<double>(1.0,3.0), complex<double>(3.0,-2.0),
        complex<double>(2.0,1.0), complex<double>(13.0,6.0)},
    {complex<double>(7.0,-2.0), complex<double>(2.0,7.0),
        complex<double>(1.0,5.0), complex<double>(-2.0,8.0)},
    {complex<double>(9.0,9.0), complex<double>(15.0,-3.0),
        complex<double>(3.0,15.0), complex<double>(-2.0,1.0)},
    {complex<double>(-2.0,-2.0), complex<double>(-2.0,-2.0),
        complex<double>(11.0,7.0), complex<double>(5.0,6.0)}
};

matrix<complex<double> > mca(&ca[0][0], 4, 4);

complex<double> cb[4] =
{
    complex<double>(2.0,1.0), complex<double>(7.0,2.0),
    complex<double>(3.0,-2.0), complex<double>(9.0,3.0)
```

```
    };

    valarray<complex<double> > vcb(cb,4);

    if(LE_TotalChoiceGauss(mca, vcb) ! = 0)
    {
        for(i = 0; i < 4; i++)     //输出复数解
        {
            cout << "cx(" << i << ") = " ;
            ValarrayPrint(vcb, i);
        }
    }
    cout<<endl;
}
```

8. 示例结果

```
x(0) =        1.040577
x(1) =        0.987051
x(2) =        0.935040
x(3) =        0.881282

cx(0) = (0.067823,0.070782)
cx(1) = ( - 0.162341, - 0.761294)
cx(2) = (0.598524, - 0.437131)
cx(3) = (0.246456,0.113996)
```

7.2.2 全选主元高斯—约当消元法

1. 功能

用全选主元高斯—约当(Gauss-Jordan) 消元法同时求解系数矩阵相同而右端具有 m 组常数向量的 n 阶线性方程组 $\boldsymbol{AX}=\boldsymbol{B}$。其中

$$\boldsymbol{A}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix},\quad \boldsymbol{X}=\begin{bmatrix} x_{00} & x_{01} & \cdots & x_{0,m-1} \\ x_{10} & x_{11} & \cdots & x_{1,m-1} \\ \vdots & \vdots & & \vdots \\ x_{n-1,0} & x_{n-1,1} & \vdots & x_{n-1,m-1} \end{bmatrix}$$

$$\boldsymbol{B}=\begin{bmatrix} b_{00} & b_{01} & \cdots & b_{0,m-1} \\ b_{10} & b_{11} & \cdots & b_{1,m-1} \\ \vdots & \vdots & & \vdots \\ b_{n-1,0} & b_{n-1,1} & \cdots & b_{n-1,m-1} \end{bmatrix}$$

2. 算法

全选主元高斯—约当消元法的步骤如下：

对于从 0 到 n－1 作如下变换：

(1) 全选主元

从系数矩阵 $\boldsymbol{A}$ 的第 k 行、第 k 列开始的右下角子阵中选取绝对值最大的元素，并通过行交

换与列交换将它交换到主元素的位置上。

(2) 归一化

$$a_{kj}/a_{kk} \Rightarrow a_{kj}, j = k+1, \cdots, n-1$$
$$b_{kj}/a_{kk} \Rightarrow b_{kj}, j = 0, 1 \cdots, m-1$$

(3) 消元

$$a_{ij} - a_{ik}a_{kj} \Rightarrow a_{ij}, j = k+1, \cdots n-1$$
$$i = 0, 1, \cdots, n-1, i \neq k$$
$$b_{ij} - b_{ik}b_{kj} \Rightarrow b_{ij}, j = 0, 1, \cdots, m-1$$
$$i = 0, 1, \cdots, n-1, i \neq k$$

最后对 $\boldsymbol{B}$ 进行恢复，则 $\boldsymbol{B}$ 中的每一列即为每一个方程组的解。

3. 函数原型

```
//全选主元高斯一约当消元法
template <class _Ty>
int LE_TotalChoiceGaussJordan(matrix<_Ty>& a, matrix<_Ty>& b);
```

4. 参数

a:matrix<_Ty>类型引用，体积为 $n \times n$。存放方程组的系数矩阵，返回时将被破坏。

b:matrix<_Ty>类型引用，体积为 $n \times m$。存放方程组右端的 m 组常数向量(一列为一组)；返回时存放 m 组解向量(一列为一组)。

5. 返回值

int 类型。

0:矩阵为奇异阵。

1:程序正常结束。

6. 函数说明

LE_TotalChoiceGaussJordan()全选主元高斯一约当(Gauss_Jordan)消元法的系数矩阵和右边向量可以是实数，也可以是复数。若是复数，解也是复数。

7. 示例

分别求解 4 阶实方程组 $\boldsymbol{AX}=\boldsymbol{B}$ 和复方程组 $\boldsymbol{CX}=\boldsymbol{D}$，其中

$$\boldsymbol{A} = \begin{bmatrix} 1 & 3 & 2 & 13 \\ 7 & 2 & 1 & -2 \\ 9 & 15 & 3 & -2 \\ -2 & -2 & 11 & 5 \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} 9 & 0 \\ 6 & 4 \\ 11 & 7 \\ -2 & 1 \end{bmatrix}$$

$$\boldsymbol{C} = \begin{bmatrix} (1.0,3.0) & (3.0,-2.0) & (2.0,1.0) & (13.0,6.0) \\ (7.0,-2.0) & (2.0,7.0) & (1.0,5.0) & (-2.0,8.0) \\ (9.0,9.0) & (15.0,-3.0) & (3.0,15.0) & (-2.0,1.0) \\ (-2.0,-2.0) & (-2.0,-2.0) & (11.0,7.0) & (5.0,6.0) \end{bmatrix},$$

$$\boldsymbol{D} = \begin{bmatrix} (2.0,1.0) & (-2.0,3.0) \\ (7.0,2.0) & (3.0,7.0) \\ (3.0,-2.0) & (2.0,9.0) \\ (9.0,3.0) & (1.0,2.0) \end{bmatrix}$$

示例程序如下：

```
//LE_TotalChoiceGaussJordan.cpp     全选主元高斯一约当消元法

#include <iostream>                 //输入输出流头文件
#include "LinearEquation.h"         //线性方程(组)求解头文件

void main()
{
  int i;
    double a[4][4] =                //实系数矩阵
  {
      {1.0, 3.0, 2.0, 13.0},
      {7.0, 2.0, 1.0, -2.0},
      {9.0, 15.0, 3.0, -2.0},
      {-2.0, -2.0, 11.0, 5.0}
  };

  double b[4][2] =
  {
      {9.0, 0.0},
      {6.0, 4.0},
      {11.0, 7.0},
      {-2.0, -1.0}
  };

  matrix<double> aa(&a[0][0], 4, 4);
  matrix<double> bb(&b[0][0], 4, 2);

  cout.setf(ios::fixed);  //输出数据为定点法
  cout.precision(6);      //精度6位

  if(LE_TotalChoiceGaussJordan(aa, bb) != 0)
  {
      for(i = 0; i < 4; i++)        //输出实数解
      {
        cout << "x(" << i << ") = ";
        MatrixLinePrint(bb, i);
      }
  }
  cout<<endl;

  complex<double> caa[4][4] =     //复系数矩阵
  {
      {complex<double>(1.0,3.0), complex<double>(3.0,-2.0),
```

```
        complex<double>(2.0,1.0), complex<double>(13.0,6.0)},
        {complex<double>(7.0,-2.0), complex<double>(2.0,7.0),
        complex<double>(1.0,5.0), complex<double>(-2.0,8.0)},
        {complex<double>(9.0,9.0), complex<double>(15.0,-3.0),
        complex<double>(3.0,15.0), complex<double>(-2.0,1.0)},
        {complex<double>(-2.0,-2.0), complex<double>(-2.0,-2.0),
        complex<double>(11.0,7.0), complex<double>(5.0,6.0)}
    };

    matrix<complex<double> > mcaa(&caa[0][0], 4, 4);

    complex<double> cbb[4][2] =
    {
        {complex<double>(2.0, 1.0), complex<double>(-2.0, 3.0)},
        {complex<double>(7.0, 2.0), complex<double>(3.0, 7.0)},
        {complex<double>(3.0,-2.0),complex<double>(2.0, 9.0)},
        {complex<double>(9.0, 3.0),   complex<double>(1.0, 2.0)}
    };

    matrix<complex<double> > mcbb(&cbb[0][0], 4, 2);


    if(LE_TotalChoiceGaussJordan(mcaa, mcbb) != 0)
    {
        for(i = 0; i < 4; i++)       //输出复数解
        {
            cout << "cx(" << i << ") = ";
            MatrixLinePrint(mcbb, i);
        }
    }
    cout << endl;
}
```

8. 示例结果

```
x(0) =          0.980745          0.497931
x(1) =          0.267982          0.144494
x(2) =         -0.222629          0.062858
x(3) =          0.589274         -0.081318

cx(0) = (0.067823,0.070782) (0.251179,0.581012)
cx(1) = (-0.162341,-0.761294) (0.402452,-0.143601)
cx(2) = (0.598524,-0.437131) (0.335659,0.103438)
cx(3) = (0.246456,0.113996) (-0.057554,0.207996)
```

7.2.3 三对角方程组的追赶法

1. 功能

用追赶法求解 n 阶三对角线方程组 $\boldsymbol{AX}=\boldsymbol{D}$。其中

$$\boldsymbol{A}=\begin{bmatrix} a_{00} & a_{01} & & & & \\ a_{10} & a_{11} & a_{12} & & 0 & \\ & a_{21} & a_{22} & a_{23} & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & a_{n-2,n-3} & a_{n-2,n-2} & a_{n-2,n-1} \\ & & & & a_{n-1,n-2} & a_{n-1,n-1} \end{bmatrix}$$

$$\boldsymbol{X}=(x_0,x_1,x_2\cdots x_{n-2},x_{n-1})^T$$

$$\boldsymbol{D}=(d_0,d_1,d_2,\cdots,d_{n-2},d_{n-1})^T$$

2. 算法

追赶法的本质是没有选主元的高斯(Gauss)消元法，只是在计算过程中考虑了三对角矩阵的特点，对于绝大部分的零元素不再作处理。其步骤如下。

(1) 对于 $k=0,1,\cdots,n-2$ 作归一化及消元处理。即

$$a_{k,k+1}/a_{kk}\Rightarrow a_{k,k+1}$$

$$d_k/a_{kk}\Rightarrow d_k$$

$$a_{k+1,k+1}-a_{k+1,k}a_{k,k+1}\Rightarrow a_{k+1,k+1}$$

$$d_{k+1}-a_{k+1,k}d_k\Rightarrow d_{k+1}$$

(2) 进行回代。即

$$d_{n-1}/a_{n-1,n-1}\Rightarrow x_{n-1}$$

$$d_k-a_{k,k+1}x_{k+1}\Rightarrow x_k,\quad k=n-2,\cdots,1,0$$

在本函数中，为便于三对角线矩阵的输入，用一个一维数组 $B(0:3n-3)$ 以行为主存放三对角线矩阵 $\boldsymbol{A}$ 中的三条对角线上的非零元素。其中 n 为矩阵 $\boldsymbol{A}$ 的阶数。显然，三对角线矩阵 $\boldsymbol{A}$ 与一维数组 B 之间有如下关系：

$$\boldsymbol{A}(i,j)=\begin{cases} B[2i+j], & i-1\leqslant j\leqslant i+1 \\ 0, & \text{其他} \end{cases}$$

其中 $i,j=0,1,\cdots,n-1$。

根据以上关系，可以得到如下相应的计算过程。

(1) 对于 $k=0,1,\cdots,n-2$ 作归一化消元处理

$$B(3k+1)/B(3k)\Rightarrow B(3k+1)$$

$$d_k/B(3k)\Rightarrow d_k$$

$$B(3k+3)-B(3k+2)B(3k+1)\Rightarrow B(3k+3)$$

$$d_{k+1}-B(3k+2)d_k\Rightarrow d_{k+1}$$

(2) 进行回代

$$d_{n-1}/B(3n-3)\Rightarrow d_{n-1}$$

$$d_k-B(3k+1)d_{k+1}\Rightarrow d_k,k=n-2,\cdots 1,0$$

最后的计算结果仍放在一维数组 D 中。

由于追赶法本质上是没有选主元的高斯消元法，因此，只有当三对角矩阵满足下列条件：

$$|a_{00}|>|a_{01}|$$
$$|a_{ii}|\geqslant|a_{i,i-1}|+|a_{i,i+1}|,\quad i=1,2\cdots,n-2$$
$$|a_{n-1,n-1}|>|a_{n-1,n-2}|$$

时，追赶法的计算过程才不会出现中间结果数量级的巨大增长和舍入误差的严重积累。

3. 函数原型

```
//三对角方程组的追赶法
template <class _Ty>
int LE_TridiagonalEquationGauss(valarray<_Ty>& b, valarray<_Ty>& d);
```

4. 参数

b：valarray<_Ty>类型数组，长度为 m。以行为主，存放三对角线矩阵中三对角线上的元素，即在 b 中依次存放元素：

$$a_{00},a_{01},a_{10},a_{11},a_{12},a_{21},a_{22},a_{23},\cdots,a_{n-1,n-2},a_{n-1,n-1}$$

此数组在计算过程中要被破坏。

d：valarray<_Ty>类型数组，长度为 n。存放方程组右端的常数向量；返回时存放方程组的解。

5. 返回值

int 类型。

<0：表示 m 的值不正确，输出“Error!”。

0：表示程序工作失败(如主元绝对值太小或为 0)，输出“Fail!”。

>0：运算成功，正常返回。

6. 函数说明

m 与 n 的关系是 $m=(3\times n-2)$，若不对，程序将输出错误提示“Error!”，并中止程序的运行。

7. 示例

求解 5 阶三对角线方程组。

$$\begin{bmatrix}13 & 12 & & & \\ 11 & 10 & 9 & & \\ & 8 & 7 & 6 & \\ & & 5 & 4 & 3 \\ & & & 2 & 1\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\\x_4\\x_5\end{bmatrix}=\begin{bmatrix}3\\0\\-2\\6\\8\end{bmatrix}$$

其中 $n=5,m=13,b=(13,12,11,10,9,8,7,6,5,4,3,2,1)$。

示例程序如下：

```
//LE_TridiagonalEquationGauss.cpp     求解三对角线方程组的追赶法

#include <iostream>                   //输入输出流头文件
#include "LinearEquation.h"           //线性方程(组)求解头文件

void main()
{
  int i;
```

```
    double b[13] =
  {
      13.0, 12.0,
          11.0, 10.0, 9.0,
              8.0, 7.0, 6.0,
                  5.0, 4.0, 3.0,
                          2.0, 1.0
  };
    double d[5] = {3.0, 0.0, -2.0, 6.0, 8.0};

  valarray<double> bb(b,13);
  valarray<double> dd(d,5);

   if(LE_TridiagonalEquationGauss(bb, dd) > 0)
  {
    for (i = 0; i < 5; i++)
      cout << "x(" << i << ") = " << dd[i] << endl;
  }
  cout << endl;
}
```

8. 示例结果

```
x(0) = 5.71837
x(1) = -5.9449
x(2) = -0.383673
x(3) = 8.04082
x(4) = -8.08163
```

7.2.4 一般带型方程组求解

1. 功能

用列选主元高斯(Gauss)消元法求解右端具有 m 组常数向量的 n 阶一般带型方程组 $\boldsymbol{AX}=\boldsymbol{D}$。其中 $\boldsymbol{A}$ 为 n 阶带型矩阵，其元素满足

$$a_{ij}\begin{cases}\neq 0 & i-l\leqslant j\leqslant i+l\\ =0 & \text{其他}\end{cases}$$

即

$$\boldsymbol{A}=\begin{bmatrix}
a_{00} & \cdots & a_{0l} & & & & \\
\vdots & & \vdots & \ddots & & & 0 \\
a_{l0} & \cdots & a_{ll} & \cdots & a_{l,2l} & & \\
 & \ddots & & \ddots & & \ddots & \\
 & a_{n-l-1,n-2l-1} & \cdots & a_{n-l-1,n-l-1} & \cdots & a_{n-l-1,n-1} \\
0 & \ddots & & \vdots & & \vdots \\
 & & & a_{n-1,n-l-1} & \cdots & a_{n-1,n-1}
\end{bmatrix}$$

l 称为半带宽，带宽为 $il=2l+1$。

$$\boldsymbol{D}=\begin{bmatrix} d_{00} & \cdots & d_{0,m-1} \\ d_{10} & \cdots & d_{1,m-1} \\ \vdots & & \vdots \\ d_{n-1,0} & \cdots & d_{n-1,m-1} \end{bmatrix}$$

2. 算法

对此方程组的求解采用列选主高斯消元法。由于系数矩阵中非零元素是带状的，带区外均为零元素，因此，在求解过程中可以只考虑带区内的非零元素，这样可以减少许多计算工作量。同时，对于带型矩阵也只需存储带区内的非零元素，可以用一个 n 行、$(2l+1)$列的二维数组 $\boldsymbol{B}$ 来表示。为了在消元过程中不在带区外产生新的非零元素，对于带型矩阵 $\boldsymbol{A}$ 中前$l+1$行与最后$l+1$行的元素均以左边对齐顺序放在数组 $\boldsymbol{B}$ 的相应行，而最右边的空余部分均填入零。即带型矩阵 $\boldsymbol{A}$ 用二维数组 $\boldsymbol{B}$ 表示的格式为

$$\boldsymbol{B}=\begin{bmatrix} b_{00} & \cdots & b_{01} & \cdots & b_{0,2l} \\ \vdots & & \vdots & & \vdots \\ b_{10} & \cdots & b_{11} & \cdots & b_{1,2l} \\ \vdots & & \vdots & & \vdots \\ b_{n-l-1,0} & \cdots & b_{n-l-1,l} & \cdots & b_{n-l-1,2l} \\ \vdots & & \vdots & & \vdots \\ b_{n-1,0} & \cdots & b_{n-1,l} & \cdots & b_{n-1,2l} \end{bmatrix}$$

$$=\begin{bmatrix} a_{00} & \cdots & a_{0l} & & 0 \\ \vdots & \ddots & & \ddots & \\ a_{10} & \cdots & a_{1l} & \cdots & a_{1,2l} \\ \vdots & & \vdots & & \vdots \\ a_{n-l-1,n-2l-1} & \cdots & b_{n-l-1,n-l-1} & \cdots & b_{n-l-1,n-1} \\ \vdots & \iddots & & \iddots & 0 \\ a_{n-1,n-l-1} & \cdots & a_{n-1,n-1} & & \end{bmatrix}$$

整个求解过程(包括列选主元)均在二维数组 $\boldsymbol{B}$ 中进行。

由于本函数采用的是列选主元高斯消元法，因此，对于大型带矩阵、且带宽较大的方程组，其数值计算有可能是不稳定的。

3. 函数原型

```
//一般带型方程组求解
template <class _Ty>
int LE_StrapEquationGauss(matrix<_Ty>& b, matrix<_Ty>& d, int l, int il);
```

4. 参数

b:matrix<_Ty>类型引用，体积为 $n\times il$ 。存放带型矩阵 $\boldsymbol{A}$ 中带区内的元素，存放格式见本节的方法说明。此数组在本函数中要被破坏。

d:matrix<_Ty>类型引用，体积为 $n\times m$ 。存放方程组右端的 m 组常数向量；返回时存放方程组的 m 组解。

l:整型变量。系数矩阵的半带宽。

il:整型变量。系数矩阵的带宽，应为 $il=2l+1$ 。

5. 返回值

int 类型。

<0:表示参数中半带宽 l 与带宽 il 的关系不对,输出信息"fail"。

0:表示系数矩阵 A 奇异,输出信息"fail"。

>0: 运算成功,正常返回。

6. 函数说明

il 与 l 的关系是 $il=(2*l+1)$,若不对,程序将输出错误提示"fail",并中止程序的运行。

7. 示例

求解 8 阶五对角线方程组 $AX=D$。其中

$$A=\begin{bmatrix} 3 & -4 & 1 & & & & & \\ -2 & -5 & 6 & 1 & & & & \\ 1 & 3 & -1 & 2 & -3 & & & \\ & 2 & 5 & -5 & 6 & -1 & & \\ & & -3 & 1 & -1 & 2 & 5 & \\ & & & 6 & 1 & -3 & 2 & -9 \\ & & & & -4 & 1 & -1 & 2 \\ & & & & & 5 & 1 & -7 \end{bmatrix}$$

$$D=\begin{bmatrix} 13 & 29 & -13 \\ -6 & 17 & -21 \\ -31 & -6 & 4 \\ 64 & 3 & 16 \\ -20 & 1 & -5 \\ -22 & -41 & 56 \\ -29 & 10 & -21 \\ 7 & -24 & 20 \end{bmatrix}$$

与带型矩阵 A 所对应的二维数组 B 为

$$B=\begin{bmatrix} 3 & -4 & 1 & 0 & 0 \\ -2 & -5 & 6 & 1 & 0 \\ 1 & 3 & -1 & 2 & -3 \\ 2 & 5 & -5 & 6 & -1 \\ -3 & 1 & -1 & 2 & -5 \\ 6 & 1 & -3 & 2 & -9 \\ -4 & 1 & -1 & 2 & 0 \\ 5 & 1 & -7 & 0 & 0 \end{bmatrix}$$

在本问题中,$n=8$,半带宽 $l=2$,带宽 $il=5$,$m=3$。

示例程序如下:

```
//LE_StrapEquationGauss().cpp      一般带型方程组的求解

#include <iostream>                //输入输出流头文件
#include "LinearEquation.h"        //线性方程(组)求解头文件
```

```
void main()
{
  double b[8][5] =
  {
      { 3.0,  -4.0,  1.0,  0.0,  0.0},
      {-2.0, -5.0,  6.0,  1.0,  0.0},
      {1.0,   3.0,   -1.0, 2.0, -3.0},
      { 2.0,  5.0,   -5.0, 6.0, -1.0},
      {-3.0,  1.0,   -1.0, 2.0, -5.0},
      { 6.0,  1.0,   -3.0, 2.0, -9.0},
      {-4.0,  1.0,   -1.0, 2.0,  0.0},
      {5.0,   1.0,   -7.0, 0.0,  0.0}
  };

  double d[8][3] =
  {
      { 13.0,  29.0,  -13.0},
      { -6.0, 17.0,  -21.0},
      {-31.0, -6.0,  4.0},
      { 64.0,  3.0,   16.0},
      {-20.0, 1.0,   -5.0},
      {-22.0,  -41.0, 56.0},
      {-29.0,  10.0,   -21.0},
      { 7.0,    -24.0, 20.0}
  };

  matrix<double> bb(&b[0][0], 8, 5);
  matrix<double> dd(&d[0][0], 8, 3);

    if(LE_StrapEquationGauss(bb, dd, 2, 5) > 0)
  {
      cout.width(12);
      cout << "Group 1";
      cout.width(16);
      cout << "Group 2";
      cout.width(16);
      cout << "Group 3" << endl << endl;
      MatrixLinePrint(dd);
  }
  cout << endl;
}
```

8. 示例结果

```
    Group 1          Group 2          Group 3
```

3	5	-1.27676e-015
-1	-3	3
8.32667e-016	2	-1
-5	-2.66454e-015	-1.77636e-015
7	-8.88178e-016	2
1	1	-3
2	-1	-8.88178e-016
1.51992e-016	4	-5

本方程组的准确解为：

第一组解	第二组解	第三组解
$x_0=3.0$	$x_0=5.0$	$x_0=0.0$
$x_1=1.0$	$x_1=-3.0$	$x_1=3.0$
$x_2=0.0$	$x_2=2.0$	$x_2=-1.0$
$x_3=-5.0$	$x_3=0.0$	$x_3=0.0$
$x_4=7.0$	$x_4=0.0$	$x_4=2.0$
$x_5=1.0$	$x_5=1.0$	$x_5=-3.0$
$x_6=2.0$	$x_6=-1.0$	$x_6=0.0$
$x_7=0.0$	$x_7=4.0$	$x_7=-5.0$

9. 示例说明

示例运算所得结果中的0元素不是严格的数值0，这是电子数字计算机进行数值计算本身性质所决定的。

7.2.5 对称方程组的分解法

1. 功能

用分解法求解系数矩阵对称，且右端具有 m 组常数的线性方程组 $\boldsymbol{AX}=\boldsymbol{C}$。其中 $\boldsymbol{A}$ 为 n 阶对称矩阵，$\boldsymbol{C}$ 为

$$\boldsymbol{C}=\begin{bmatrix} c_{00} & c_{01} & \cdots & c_{0,m-1} \\ c_{10} & c_{11} & \cdots & c_{1,m-1} \\ \vdots & \vdots & & \vdots \\ c_{n-1,0} & c_{n-1,1} & \cdots & c_{n-1,m-1} \end{bmatrix}$$

2. 算法

对称矩阵 $\boldsymbol{A}$ 可以分解为一个下三角矩阵 $\boldsymbol{L}$、一个对角线矩阵 $\boldsymbol{D}$ 和一个上三角矩阵 $\boldsymbol{L}^{\mathrm{T}}$ 的乘积，即 $\boldsymbol{A}=\boldsymbol{LDL}^{\mathrm{T}}$，其中：

$$\boldsymbol{L}=\begin{bmatrix} 1 & & & 0 \\ l_{1,0} & 1 & & \\ \vdots & \vdots & \ddots & \\ l_{n-1,0} & l_{n-1,1} & \cdots & 1 \end{bmatrix},\quad \boldsymbol{D}=\begin{bmatrix} d_{00} & & & \\ & d_{11} & & 0 \\ & & \ddots & \\ & 0 & & d_{n-1,m-1} \end{bmatrix}$$

矩阵 L 和 D 中的各元素由下列计算公式确定：

$$d_{00} = a_{00}$$

$$d_{ii} = a_{ii} - \sum_{k=0}^{i-1} l_{ik}^2 d_{kk}$$

$$l_{ij} = (a_{ij} - \sum_{k=0}^{j-1} l_{ik} l_{jk} d_{kk}) / d_{jj}, j < i$$

$$l_{ij} = 0, j > i$$

对于方程组 $AX=B$ 来说，当 L 和 D 确定后，令 $DL^{\mathrm{T}}X=Y$，则可由回代过程求解方程组 $LY=B$ 而得到 Y，再由方程组 $DL^{\mathrm{T}}X=Y$ 解出 X。其计算公式如下：

$$y_0 = b_0$$

$$y_i = b_i - \sum_{k=0}^{i-1} l_{ik} y_k, i > 0$$

$$x_{n-1} = y_{n-1} / d_{n-1,n-1}$$

$$x_i = (y_i - \sum_{k=i+1}^{n-1} d_{ii} l_{ki} x_k) / d_{ii}$$

3. 函数原型

```
//对称方程组的分解法
template <class _Ty>
int LE_SymmetryEquation(matrix<_Ty>& a, matrix<_Ty>& c);
```

4. 参数

a：matrix<_Ty>类型引用，体积为 $n \times n$。存放方程组的系数矩阵(应为对称矩阵)。此数组在本函数中要被破坏。

c：matrix<_Ty>类型引用，体积为 $n \times m$。存放方程组右端的 m 组常数向量；返回时存放方程组的 m 组解。

5. 返回值

int 类型。

－1：表示矩阵 a 不对称。

－2：主元为 0。

＋1：运算成功，正常返回。

6. 函数说明

程序中调用了判别矩阵 a 是否对称的函数 MatrixSymmetry()，请参阅第 5 章 5.3.2 节。

7. 示例

求解 5 阶取称方程组 $AX=C$。其中

$$A = \begin{bmatrix} 5 & 7 & 6 & 5 & 1 \\ 7 & 10 & 8 & 7 & 2 \\ 6 & 8 & 10 & 9 & 3 \\ 5 & 7 & 9 & 10 & 4 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}, \quad C = \begin{bmatrix} 24 & 96 \\ 34 & 136 \\ 36 & 144 \\ 35 & 140 \\ 15 & 60 \end{bmatrix}$$

示例程序如下：

```
//LE_SymmetryEquation.cpp      求解对称方程组的分解法
```

```
#include <iostream>          //输入输出流头文件
#include "LinearEquation.h"  //线性方程(组)求解头文件

void main()
{
  int i;

  double a[5][5] =
  {
      {5.0, 7.0, 6.0, 5.0, 1.0},
      {7.0, 10.0, 8.0, 7.0, 2.0},
      {6.0, 8.0, 10.0, 9.0, 3.0},
      {5.0, 7.0, 9.0, 10.0, 4.0},
      {1.0, 2.0, 3.0, 4.0, 5.0}
  };

    double c[5][2] =
  {
      {24.0, 96.0},
      {34.0, 136.0},
      {36.0, 144.0},
      {35.0, 140.0},
      {15.0, 60.0}
  };

  matrix<double> aa(&a[0][0], 5, 5);
  matrix<double> cc(&c[0][0], 5, 2);

  cout.setf(ios::fixed); //输出数据为定点法
  cout.precision(6); //精度 6 位

  if(LE_SymmetryEquation(aa, cc) > 0)
  {
    cout.width(22);
    cout << "Group 1";
    cout.width(16);
    cout << "Group 2" << endl << endl;

    for(i = 0; i < 5; i++)
    {
      cout << "x(" << i << ") = ";
      MatrixLinePrint(cc, i);
    }
```

```
    }
    cout << endl;
}
```

8. 示例结果

```
                Group 1      Group 2

x(0) =      1.000000     4.000000
x(1) =      1.000000     4.000000
x(2) =      1.000000     4.000000
x(3) =      1.000000     4.000000
x(4) =      1.000000     4.000000
```

本方程组的准确解为：

第一组解　$x_0=x_1=x_2=x_3=x_4=1$

第二组解　$x_0=x_1=x_2=x_3=x_4=4$

7.2.6　对称正定方程组的平方根法

1. 功能

用乔里斯基(Cholesky)分解法(即平方根法)求解系数矩阵为对称正定、且右端具有 m 组常数向量的 n 阶线性方程组 $\boldsymbol{AX}=\boldsymbol{D}$。

2. 算法

当系数矩阵 $\boldsymbol{A}$ 为对称正定时，可以唯一地分解为 $\boldsymbol{A}=\boldsymbol{U}^{\mathrm{T}}\boldsymbol{U}$，其中 $\boldsymbol{U}$ 为上三角矩阵，即

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}$$

$$= \begin{bmatrix} u_{00} & & & \\ u_{10} & u_{11} & & 0 \\ \vdots & \vdots & \ddots & \\ u_{n-1,0} & u_{n-1,1} & \cdots & u_{n-1,n-1} \end{bmatrix} \begin{bmatrix} u_{00} & u_{01} & \cdots & u_{0,n-1} \\ & u_{11} & \cdots & u_{1,n-1} \\ & 0 & \ddots & \vdots \\ & & & u_{n-1,n-1} \end{bmatrix}$$

且 $u_{ij}=u_{ji}(i,j=0,1,\cdots,n-1)$。

$\boldsymbol{U}$ 矩阵中的各元素由以下计算公式确定：

$$u_{00}=\sqrt{a_{00}}$$

$$u_{ii}=(a_{ii}-\sum_{k=0}^{i-1}u_{ki}^2)^{\frac{1}{2}},i=1,2,\cdots,n-1$$

$$u_{ij}=(a_{ij}-\sum_{k=0}^{i-1}u_{ki}u_{kj})/u_{ii},j>i$$

于是，方程组 $\boldsymbol{AX}=\boldsymbol{B}$ 的解可由下述公式计算：

$$y_i=(b_i-\sum_{k=0}^{i-1}u_{ki}y_k)/u_{ii}$$

$$x_i = (y_i - \sum_{k=i+1}^{n-1} u_{ik}x_k)/u_{ii}$$

3. 函数原型

```
//对称正定方程组的平方根法
template <class _Ty>
int LE_SymmetryRegularEuationSquareRoot(matrix<_Ty>& a, matrix<_Ty>& d);
```

4. 参数

a:matrix<_Ty>类型引用,体积为 $n \times n$。存放系数矩阵(应为对称正定矩阵);返回时,其上三角部分存放分解后的 U 矩阵,即

$$U = \begin{bmatrix} u_{00} & u_{01} & \cdots & u_{0,n-1} \\ & u_{11} & \cdots & u_{1,n-1} \\ & & \ddots & \cdots \\ & & & u_{n-1,n-1} \end{bmatrix}$$

d:matrix<_Ty>类型引用,体积为 $n \times m$。存放方程组右端 m 组常数向量;返回时,存放方程组的 m 组解。

5. 返回值

int 类型。

-1:表示矩阵 a 不是对称正定阵。

+1:运算成功,正常返回。

6. 函数说明

程序中调用了判别矩阵 a 是否对称正定的函数 MatrixSymmetryRegular(),其第二个参数取大于 0 的正整数,判别矩阵 a 是否对称且正定。当返回值为 2 时,表示矩阵 a 为对称正定阵。在该函数运行过程中会输出受判别矩阵的顺序主子式数值。该函数细节请参阅第 5 章 5.3.3 节。

7. 示例

求解 4 阶对称正定方程组 $AX=D$。其中

$$A = \begin{bmatrix} 5 & 7 & 6 & 5 \\ 7 & 10 & 8 & 7 \\ 6 & 8 & 10 & 9 \\ 5 & 7 & 9 & 10 \end{bmatrix}, \quad D = \begin{bmatrix} 23 & 92 \\ 32 & 128 \\ 33 & 132 \\ 31 & 124 \end{bmatrix}$$

示例程序如下:

```
//LE_SymmetryRegularEuationSquareRoot.cpp  求解对称正定方程组的平方根法

#include <iostream>                    //输入输出流头文件
#include "LinearEquation.h"            //线性方程(组)求解头文件

void main()
{
  double a[4][4] =
  {
      {5.0, 7.0, 6.0, 5.0},
      {7.0, 10.0, 8.0, 7.0},
```

```
        {6.0, 8.0, 10.0, 9.0},
        {5.0, 7.0, 9.0, 10.0}
    };

    double d[4][2] =
    {
        {23.0, 92.0},
        {32.0, 128.0},
        {33.0, 132.0},
        {31.0, 124.0}
    };

    matrix<double> aa(&a[0][0], 4, 4);
    matrix<double> dd(&d[0][0], 4, 2);

    if(LE_SymmetryRegularEuationSquareRoot(aa, dd) > 0)
    {
        cout << endl;
        cout.width(22);
        cout << "Group 1";
        cout.width(16);
        cout << "Group 2" << endl << endl;

        for(int i = 0; i < 4; i ++)
        {
            cout << " x(" << i << ") = ";
            MatrixLinePrint(dd, i);
        }
     }
     cout << endl;
}
```

8. 示例结果

```
K = 1  Determinant = 5.000000
K = 2  Determinant = 1.000000
K = 3  Determinant = 2.000000
K = 4  Determinant = 1.000000

                Group 1          Group 2

x(0) =          1.000000         4.000000
x(1) =          1.000000         4.000000
x(2) =          1.000000         4.000000
x(3) =          1.000000         4.000000
```

本方程组的准确解为：

第一组解　$x_0=x_1=x_2=x_3=1$

第二组解　$x_0=x_1=x_2=x_3=4$

7.2.7　大型稀疏方程组全选主元高斯—约当法

1. 功能

用全选主元高斯—约当(Gauss-Jordan)消元法求解系数矩阵为稀疏矩阵的大型方程组 $\boldsymbol{AX}=\boldsymbol{B}$。

2. 算法

基本方法同本章 7.2.2 节，只是在消元过程中避免了对零元素的运算，从而可以大大减少运算次数。

3. 函数原型

```
//大型稀疏方程组全选主元高斯—约当(Gauss-Jordan)消元法
template <class _Ty>
int LE_SparseEuationTotalChoiceGaussJordan(matrix<_Ty>& a, valarray<_Ty>& b);
```

4. 参数

a：matrix<_Ty>类型引用，体积为 $n\times n$，存放方程组的系数矩阵。此数组在本函数中要被破坏。

b：valarray<_Ty>类型引用，长度 n 为，存放方程组右端的常数向量，返回时存放方程组的解。

5. 返回值

int 类型。

0：表示矩阵 a 奇异，并输出信息“fail”。

1：运算成功，正常返回。

6. 函数说明

本函数未考虑稀疏矩阵的压缩存储问题。

7. 示例

求解放阶稀疏方程组 $\boldsymbol{AX}=\boldsymbol{B}$。其中

$$\boldsymbol{A}=\begin{bmatrix}0&0&-1&0&0&0&2&0\\0&6&0&0&0&-6&0&0\\0&0&0&2&0&0&0&-4\\3&0&0&0&-2&0&1&0\\0&0&6&0&0&0&5&0\\1&0&0&0&-3&0&0&2\\0&4&0&-1&0&0&0&0\\0&0&1&0&-1&0&0&-2\end{bmatrix}$$

$$\boldsymbol{B}=(4,6,-8,-2,27,-9,2,-4)^{\mathrm{T}}$$

示例程序如下：

```
//LE_SparseEuationTotalChoiceGaussJordan.cpp  求解大型稀疏方程组的全选主元高斯—约当消元法

#include <iostream>                    //输入输出流头文件
#include "LinearEquation.h"            //线性方程(组)求解头文件
```

```
void main()
{
  int i;

  double a[8][8] =
  {
      {0.0, 0.0, -1.0,  0.0,  0.0,   0.0,  2.0,  0.0},
      {0.0, 6.0,  0.0,  0.0,  0.0,  -6.0,  0.0,  0.0},
      {0.0, 0.0,  0.0,  2.0,  0.0,   0.0,  0.0, -4.0},
      {3.0, 0.0,  0.0,  0.0, -2.0,   0.0,  1.0,  0.0},
      {0.0, 0.0,  6.0,  0.0,  0.0,   0.0,  5.0,  0.0},
      {1.0, 0.0,  0.0,  0.0, -3.0,   0.0,  0.0,  2.0},
      {0.0, 4.0,  0.0, -1.0,  0.0,   0.0,  0.0,  0.0},
      {0.0, 0.0,  1.0,  0.0, -1.0,   0.0,  0.0, -2.0}
  };

  double b[8] = {4.0, 6.0, -8.0, -2.0, 27.0, -9.0, 2.0, -4.0};

  valarray<double> bb(b,8);
  matrix<double> aa(&a[0][0], 8, 8);

  if (LE_SparseEuationTotalChoiceGaussJordan(aa, bb) ! = 0)
      for (i=0; i<8; i++)
      {
          cout << "x(" << i << ") = " << bb[i] << endl;
      }
      cout << endl;
}
```

8. 示例结果

```
x(0) = 1
x(1) = 0
x(2) = 2
x(3) = -2
x(4) = 4
x(5) = -1
x(6) = 3
x(7) = 1
```

7.2.8 托伯利兹方程组的列文逊法

1. 功能

用列文逊(Levinson)递推算法求解对称托伯利兹(Toeplitz)型方程组。

2. 算法

n 阶对称托伯利兹矩阵为如下形式的矩阵：

$$\boldsymbol{T}^{(n)}=\begin{bmatrix} t_0 & t_1 & t_2 & \cdots & t_{n-1} \\ t_1 & t_0 & t_1 & \cdots & t_{n-2} \\ t_2 & t_1 & t_0 & \cdots & t_{n-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_{n-1} & t_{n-2} & t_{n-3} & \cdots & t_0 \end{bmatrix}$$

简称 n 阶对称 T 型矩阵。

设线性方程组 $\boldsymbol{AX}=\boldsymbol{B}$ 的系数矩阵为 n 阶对称 T 型矩阵，即 $\boldsymbol{A}=\boldsymbol{T}^{(n)}$。

假设已知

$$\boldsymbol{T}^{(k)}\begin{bmatrix} y_0^{(k)} \\ \vdots \\ y_{k-2}^{(k)} \\ y_{k-1}^{(k)} \end{bmatrix}=\begin{bmatrix} 0 \\ \vdots \\ 0 \\ \alpha_{k-1} \end{bmatrix}$$

令

$$\beta_{k-1}=y_{k-1}^{(k)}t_{(k)}+y_{k-2}^{(k)}t_{k-1}+\cdots+y_0^{(k)}t_1=\sum_{j=0}^{k-1}t_{j+1}y_j^{(k)} \tag{1}$$

由于 $\boldsymbol{T}^{(k)}$ 与 $\boldsymbol{T}^{(k+1)}$ 均为对称 T 型矩阵，因此有

$$\boldsymbol{T}^{(k+1)}\begin{bmatrix} 0 \\ y_0^{(k)} \\ \vdots \\ y_{k-2}^{(k)} \\ y_{k-1}^{(k)} \end{bmatrix}=\begin{bmatrix} \beta_{k-1} \\ 0 \\ \vdots \\ 0 \\ \alpha_{k-1} \end{bmatrix} \tag{2}$$

与

$$\boldsymbol{T}^{(k+1)}\begin{bmatrix} y_{k-1}^{(k)} \\ y_{k-2}^{(k)} \\ \vdots \\ y_0^{(k)} \\ 0 \end{bmatrix}=\begin{bmatrix} \alpha_{k-1} \\ 0 \\ \vdots \\ 0 \\ \beta_{k-1} \end{bmatrix} \tag{3}$$

若将方程组(2)减去方程组(3)的 β_{k-1}/α_{k-1} 倍，并且令

$$c_{k-1}=-\beta_{k-1}/\alpha_{k-1} \tag{4}$$

则可得到如下方程组

$$\boldsymbol{T}^{(k+1)}\begin{bmatrix} c_{k-1}y_{k-1}^{(k)} \\ y_0^{(k)}+c_{k-1}y_{k-2}^{(k)} \\ \vdots \\ y_{k-2}^{(k)}+c_{k-1}y_0^{(k)} \\ y_{k-1}^{(k)} \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \alpha_{k-1}+c_{k-1}\beta_{k-1} \end{bmatrix} \tag{5}$$

若再令

$$\begin{cases} y_0^{(k+1)}=c_{k-1}y_{k-1}^{(k)} \\ y_i^{(k+1)}=y_{i-1}^{(k)}+c_{k-1}y_{k-i-1}^{(k)},\ i=1,2,\cdots,k-1 \\ y_k^{(k+1)}=y_{k-1}^{(k)} \end{cases} \tag{6}$$

及

$$\alpha_k = \alpha_{k-1} + c_{k-1}\beta_{k-1} \tag{7}$$

则方程组(5)变为

$$\boldsymbol{T}^{(k+1)}\begin{bmatrix} y_0^{(k+1)} \\ y_1^{(k+1)} \\ \vdots \\ y_{k-1}^{(k+1)} \\ y_k^{(k+1)} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \alpha_k \end{bmatrix} \tag{8}$$

显然,式(6)与(7)是递推公式。若取 $\alpha_0 = t_0$,则 $y_0^{(1)} = 1$ 。

下面再考虑方程组

$$\begin{aligned} &\boldsymbol{T}^{(k)}x^{(k)} = \boldsymbol{B}^{(k)},\ k = 2,3\cdots,n \\ &\boldsymbol{X}^{(k)} = (\boldsymbol{X}_0^{(k)},\boldsymbol{X}_2^{(k)},\cdots,\boldsymbol{X}_{k-1}^{(k)})^{\mathrm{T}} \\ &\boldsymbol{B}^{(k)} = (b_0,b_1,\cdots,b_{k-1}) \end{aligned} \tag{9}$$

现假设对于某个 k,方程组(9)已经解出,则

$$\boldsymbol{T}^{(k+1)}\begin{bmatrix} \boldsymbol{X}^{(k+1)} \\ 0 \end{bmatrix} = \begin{bmatrix} \boldsymbol{B}^{(k)} \\ \boldsymbol{q}_k \end{bmatrix} \tag{10}$$

其中

$$q_k = x_0^{(k)}t_k + \cdots + x_{k-1}^{(k)}t_1 = \sum_{j=0}^{k-1} x_j^{(k)}t_{k-j} \tag{11}$$

又因为

$$\boldsymbol{T}^{(k+1)}\boldsymbol{X}^{(k+1)} = \boldsymbol{B}^{(k+1)} \tag{12}$$

于是,方程组(12)减去(10)得

$$\boldsymbol{T}^{(k+1)}\begin{bmatrix} x_0^{(k+1)} - x_0^{(k)} \\ x_1^{(k+1)} - x_1^{(K)} \\ \vdots \\ x_{k-1}^{(k+1)} - x_{k-1}^{(k)} \\ x_k^{(k+1)} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ b_k - q_k \end{bmatrix} \tag{13}$$

在上述过程中,当 $k=1$ 时有 $x_0^{(1)} = b_0/t_0$,如果用

$$w_k = (b_k - q_k/a_k) \tag{14}$$

乘方程组(8),可以得到

$$\boldsymbol{T}^{(k+1)}\begin{bmatrix} w_k y_0^{(k+1)} \\ w_k y_1^{(k+1)} \\ \vdots \\ w_k y_{k-1}^{(k+1)} \\ w_k y_k^{(k+1)} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ b_k - q_k \end{bmatrix} \tag{15}$$

比较方程组(13)与(15),可以得到由 $\boldsymbol{X}^{(k)}$ 计算 $\boldsymbol{X}^{(k+1)}$ 的递推公式如下:

$$\begin{cases} x_i^{(k+1)} = x_i^{(k)} + w_k y_i^{(k+1)}, i = 0,1,\cdots,k-1 \\ x_k^{(k+1)} = w_k y_k^{(k+1)} \end{cases} \tag{16}$$

综上所述,由式(11)、(3)、(4)、(6)、(7)、(14)及(16),可以得到求解对称 $\boldsymbol{T}$ 型方程组的递

推算法如下。

取初值 $a_0 = t_0, y_0^{(1)} = 1, x_0^{(1)} = b_0/t_0$

对于 k 从 1 到 $n-1$，作如下计算：

$$q_k = \sum_{j=0}^{k-1} x_j^{(k)} t_{k-j}$$

$$\beta_{k-1} = \sum_{j=0}^{k-1} y_j^{(k)} t_{j+1}$$

$$c_{k-1} = -\beta_{k-1}/a_{k-1}$$

$$\begin{cases} y_0^{(k+1)} = c_{k-1} y_{k-1}^{(k)} \\ y_i^{(k+1)} = y_{i-1}^{(k)} + c_{k-1} y_{k-i-1}^{(k)}, i = 1,2,\cdots,k-1 \\ y_k^{(k+1)} = y_{k-1}^{(k)} \end{cases}$$

$$a_k = a_{k-1} + c_{k-1}\beta_{k-1}$$

$$w_k = (b_k - q_k)/a_k$$

$$\begin{cases} x_i^{(k+1)} = x_i^{(k)} + w_k y_i^{(k+1)}, i = 0,1,\cdots,k-1 \\ x_k^{(k+1)} = w_k y_k^{(k+1)} \end{cases}$$

3. 函数原型

```
//托伯利兹(Toeplitz)方程组的列文逊(Levinson)法
template <class _Ty>
int LE_ToeplitzEuationLevinson(valarray<_Ty>& t, valarray<_Ty>& b,
valarray<_Ty>& x);
```

4. 参数

t：valarray<_Ty>类型引用，长度为 n，存放对称 T 型矩阵中的元素 $t_0, t_1, \cdots, t_{n-1}$。

b：valarray<_Ty>类型引用，长度为 n，存放方程组右端的常数向量。

x：valarray<_Ty>类型引用，长度为 n，返回方程组的解。

5. 返回值

int 类型。

<0：表示程序工作失败，并输出信息“fail”。

>0：运算成功，正常返回。

6. 示例

求解 6 阶对称 T 型方程组 $AX=B$，其中

$$A = \begin{bmatrix} 6 & 5 & 4 & 3 & 2 & 1 \\ 5 & 6 & 5 & 4 & 3 & 2 \\ 4 & 5 & 6 & 5 & 4 & 3 \\ 3 & 4 & 5 & 6 & 5 & 4 \\ 2 & 3 & 4 & 5 & 6 & 5 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} = T^{(6)}$$

即 $t=(6,5,4,3,2,1)$，常数向量 $B=(11,9,9,9,13,17)^T$。

示例程序如下：

```
//LE_ToeplitzEuationLevinson().cpp  求解托伯利兹方程组的列文逊方法

#include <iostream>                    //输入输出流头文件
```

```
#include "LinearEquation.h"          //线性方程(组)求解头文件

void main()
{
  int i;
    double t[6] = {6.0, 5.0, 4.0, 3.0, 2.0, 1.0};
    double b[6] = {11.0, 9.0, 9.0, 9.0, 13.0, 17.0};

    valarray<double> xx(6);
  valarray<double> tt(t, 6);
  valarray<double> bb(b, 6);

  if (LE_ToeplitzEuationLevinson(tt, bb, xx) > 0)
    for (i = 0; i < 6; i++)
    {
        cout << "x(" << i << ") = " << xx[i] << endl;
    }

  cout << endl;
}
```

7. 示例结果

```
x(0) = 3
x(1) = -1
x(2) = 1.15957e-015
x(3) = -2
x(4) = 8.88178e-016
x(5) = 4
```

8. 示例说明

示例运算所得结果中的0元素不是严格的数值0,这是电子数字计算机进行数值计算本身性质所决定的。

7.2.9 高斯—赛德尔迭代法

1. 功能

用高斯—赛德尔(Gauss-Seidel)迭代法求解系数矩阵具有主对角线占绝对优势的线性方程组 $\boldsymbol{AX}=\boldsymbol{B}$。其中

$$\sum_{\substack{j=0\\ j\neq i}}^{n-1}|a_{ij}|<|a_{ii}|,\quad i=0,1,\cdots,n-1$$

2. 算法

高斯—赛德尔迭代格式为

$$x_i^{(k+1)}=(b_i-\sum_{j=0}^{i-1}a_{ij}x_j^{(k+1)}-\sum_{j=i+1}^{n-1}a_{ij}x_j^{(k)})/a_{ii},i=0,1,\cdots,n-1$$

其中初值取 $x_i^{(0)}=0(i=0,1,\cdots,n-1)$。

结束迭代的条件为

$$\max_{0\leqslant i\leqslant n-1}\frac{\left|x_i^{(k+1)}-x_i^{(k)}\right|}{1+\left|x_i^{(k+1)}\right|}<\varepsilon$$

其中 ε 为给定的精度要求。

3. 函数原型

```
//高斯—赛德尔迭代法
template <class _Ty>
int LE_GaussSeidelIteration(matrix<_Ty>& a, valarray<_Ty>& b, valarray<_Ty>& x, _Ty eps);
```

4. 参数

a:matrix<_Ty>类型引用,体积为 $n\times n$,存放方程组的系数矩阵。

b:valarray<_Ty>类型引用,长度 n 为,存放方程组右端的常数向量。

x:valarray<_Ty>类型引用,长度 n 为,返回时存放方程组的解。

eps:_Ty 类型,给定的精度要求。

5. 返回值

int 类型。

<0:说明系数矩阵不具有主对角线占绝对优势,还输出信息"fail"。

>0:运算成功,正常返回。

6. 函数说明

本函数求解的方程组系数矩阵需具有主对角线占绝对优势,否则求解失败。

7. 示例

用高斯—赛德尔(Gauss-Seidel)迭代法求解下列 4 阶方程组:

$$\begin{cases}7x_0+2x_1+x_2-2x_3=4\\9x_0+15x_1+3x_2-2x_3=7\\-2x_0-2x_1+11x_2+5x_3=-1\\x_0+3x_1+2x_2+13x_3=0\end{cases}$$

取 $\varepsilon=10^{-6}$。

示例程序如下:

```
//LE_GaussSeidelIteration.cpp     高斯—赛德尔迭代法

#include <iostream>               //输入输出流头文件
#include "LinearEquation.h"       //线性方程(组)求解头文件

void main()
{
  int i;
  double eps;

  double a[4][4] =
  {
      {7.0, 2.0, 1.0, -2.0},
      {9.0, 15.0, 3.0, -2.0},
      {-2.0, -2.0, 11.0, 5.0},
```

```
        {1.0, 3.0, 2.0, 13.0}
    };

    double b[4] = {4.0, 7.0, -1.0, 0.0};

    valarray<double> xx(5);
    valarray<double> bb(b, 4);
    matrix<double> aa(&a[0][0], 4, 4);

    eps = FLOATERROR;
    cout.setf(ios::fixed);  //输出数据为定点法
    cout.precision(6);      //精度 6 位

    if(LE_GaussSeidelIteration(aa, bb, xx, eps) > 0)
    {
        for(i = 0; i < 4; i++)
        {
          cout << "x(" << i << ") = " << xx[i] << endl;
        }
    }
    cout << endl;
}
```

8. 示例结果

```
x(0) = 0.497931
x(1) = 0.144494
x(2) = 0.062858
x(3) = -0.081318
```

7.2.10 对称正定方程组的共轭梯度法

1. 功能

使用共轭梯度法求解 n 阶对称正定方程组 $\boldsymbol{AX}=\boldsymbol{B}$。

2. 算法

共轭梯度法的递推计算公式如下。

取初值 $X_0=(0,0,\cdots,0)^T$，$R_0=P_0=B$。

对于 $i=0,1,\cdots,n-1$ 作如下计算：

$$a_i=\frac{(P_i,B)}{(P_i,AP_i)}$$

$$X_{i+1}=X_i+a_iP_i$$

$$R_{i+1}=B-AX_{i+1}$$

$$\beta_i=\frac{(R_{i+1},AP_i)}{(P_i,AP_i)}$$

$$P_{i+1}=R_{i+1}-\beta_iP_i$$

上述过程一直作到 $\|R_i\| < \varepsilon$ 或 $i = n-1$ 为止。

特别指出，本方法只适用于对称正定方程组。

3. 函数原型

```
//对称正定方程组的共轭梯度法
template <class _Ty>
int LE_SymmetryRegularEuationConjugateGradient(matrix<_Ty>& a,
  valarray<_Ty>& b, _Ty eps, valarray<_Ty>& x);
```

4. 参数

a：matrix<_Ty>类型引用，体积为 $n \times n$，存放对称正定矩阵 A。

b：valarray<_Ty>类型引用，长度为 n，存放方程组右端的常数向量。

eps：_Ty 类型，控制精度要求。

x：valarray<_Ty>类型引用，长度为 n，返回方程组的解向量。

5. 返回值

int 类型。

－1：表示矩阵 a 不是对称正定阵。

＋1：运算成功，正常返回。

6. 函数说明

本函数中调用了判别矩阵 a 是否对称正定的函数 MatrixSymmetryRegular()，其第二个参数取大于 0 的正整数，判别矩阵 a 是否对称且正定。当返回值为 2 时，表示矩阵 a 为对称正定阵。在该函数运行过程中会输出为判别系数矩阵 a 是否为对称正定矩阵的顺序主子式数值。关于该函数的细节请参阅第 5 章 5.3.3 节。

本函数还调用了矩阵乘法函数 MatrixMultiply()，该函数细节请参阅第 5 章 5.2.3 节。

7. 示例

用共轭梯度法求解 4 阶对称正定方程组 $\boldsymbol{AX}=\boldsymbol{B}$。其中

$$\boldsymbol{A}=\begin{bmatrix}5 & 7 & 6 & 5\\ 7 & 10 & 8 & 7\\ 6 & 8 & 10 & 9\\ 5 & 7 & 9 & 10\end{bmatrix},\quad \boldsymbol{B}=\begin{bmatrix}23\\ 32\\ 33\\ 31\end{bmatrix}$$

示例程序如下：

```
//LE_SymmetryRegularEuationConjugateGradient.cpp
//求解对称正定方程组的共轭梯度法

#include <iostream>          //输入输出流头文件
#include "LinearEquation.h"  //线性方程(组)求解头文件

void main()
{
   int i;
   double eps = FLOATERROR;
   double a[4][4] =
   {
```

```
        {5.0, 7.0, 6.0, 5.0},
        {7.0, 10.0, 8.0, 7.0},
        {6.0, 8.0, 10.0, 9.0},
        {5.0, 7.0, 9.0, 10.0}
    };

      double b[4] = {23.0, 32.0, 33.0, 31.0};

    valarray<double> xx(4);
    valarray<double> bb(b, 4);
    matrix<double> aa(&a[0][0], 4, 4);

      if(LE_SymmetryRegularEuationConjugateGradient(aa, bb, eps, xx) > 0)
    {
        for (i = 0; i<4; i++)
        {
            cout << endl << " x(" << i << ") = " << xx[i];
      }
    }

      cout << endl;
}
```

8. 示例结果

```
K = 1  Determinant = 5
K = 2  Determinant = 1
K = 3  Determinant = 2
K = 4  Determinant = 1

x(0) = 1
x(1) = 1
x(2) = 1
x(3) = 1
```

7.2.11 线性最小二乘问题的豪斯荷尔德变换法

1. 功能

用豪斯荷尔德(Householder)变换求解线性最小二乘问题。

2. 算法

设超定方程组为 $\boldsymbol{AX}=\boldsymbol{B}$,其中 $\boldsymbol{A}$ 为 $m\times n(m\geqslant n)$ 列线性无关的矩阵,$\boldsymbol{X}$ 为 n 维列向量,$\boldsymbol{B}$ 为 m 维列向量。

用豪斯荷尔德变换将 $\boldsymbol{A}$ 进行 $\boldsymbol{QR}$ 分解。即

$$\boldsymbol{A}=\boldsymbol{Q}\boldsymbol{R}$$

其中 $\boldsymbol{Q}$ 为 $m\times m$ 的正交矩阵,$\boldsymbol{R}$ 为上三角矩阵,具体分解过程参阅第 5 章 5.6.2 节。

设 $\boldsymbol{E}=\boldsymbol{B}-\boldsymbol{AX}$,用 Q^{T} 乘上式两端得

$$\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{E}=\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{B}-\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{X}=\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{B}-\boldsymbol{R}\boldsymbol{X}$$

因为 $\boldsymbol{Q}^{\mathrm{T}}$ 为正交矩阵，所以有

$$\|\boldsymbol{E}\|_2^2=\|\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{E}\|_2^2=\|\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{B}-\boldsymbol{R}\boldsymbol{X}\|_2^2$$

若令

$$\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{B}=\begin{bmatrix}\boldsymbol{C}\\ \boldsymbol{D}\end{bmatrix},\boldsymbol{R}\boldsymbol{X}=\begin{bmatrix}\boldsymbol{R}_1\\ 0\end{bmatrix}\boldsymbol{X}$$

其中 $\boldsymbol{C}$ 为 n 维向量，$\boldsymbol{D}$ 为 $m-n$ 维列向量，$\boldsymbol{R}_1$ 为 $n\times n$ 上三角方阵，0 为 $(m-n)\times n$ 的零矩阵。则有

$$\|\boldsymbol{E}\|_2^2=\|\boldsymbol{C}-\boldsymbol{R}_1\boldsymbol{X}\|_2^2+\|\boldsymbol{D}\|_2^2$$

显然，当 $\boldsymbol{X}$ 满足 $\boldsymbol{R}_1\boldsymbol{X}=\boldsymbol{C}$ 时，$\|\boldsymbol{E}\|_2^2$ 将取最小值。

由上所述，求解线性最小二乘问题 $\boldsymbol{A}\boldsymbol{X}=\boldsymbol{B}$ 的步骤如下。

(1) 对 $\boldsymbol{A}$ 进行 $\boldsymbol{QR}$ 分解，即 $\boldsymbol{A}=\boldsymbol{QR}$。其中 $\boldsymbol{Q}$ 为 $m\times m$ 的正交矩阵，R 为右上三角矩阵，且令

$$\boldsymbol{R}=\begin{bmatrix}\boldsymbol{R}_1\\ 0\end{bmatrix}$$

其中 $\boldsymbol{R}_1$ 为 $n\times n$ 的上三角方阵。

(2) 计算

$$\begin{bmatrix}\boldsymbol{C}\\ \boldsymbol{D}\end{bmatrix}=\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{B}$$

其中 $\boldsymbol{C}$ 为 n 维列向量。

(3) 利用回代求解方程组 $\boldsymbol{R}_1\boldsymbol{X}=\boldsymbol{C}$。

3. 函数原型

```
//线性最小二乘问题的豪斯荷尔德变换法
template <class _Ty>
int LE_LinearLeastSquareHouseholder(matrix<_Ty>& a, valarray<_Ty>& b,
  matrix<_Ty>& q);
```

4. 参数

a：matrix<_Ty>类型引用，体积为 $m\times n$，存放超定方程组的系数矩阵 $\boldsymbol{A}$，返回时存放 $\boldsymbol{QR}$ 分解式中的 $\boldsymbol{R}$ 矩阵。$m\geqslant n$。

b：valarray<_Ty>类型引用，长度为 m，存放方程组右端常数向量，返回时前 n 个元素($n\leqslant m$，即 $b(0),b(1),\cdots,b(n-1)$) 存放方程组的最小二乘解。

q：matrix<_Ty>类型引用，体积为 $m\times m$，返回时存放 $\boldsymbol{QR}$ 分解式中的正交矩阵 $\boldsymbol{Q}$。

5. 返回值

int 类型。

−1：表示程序工作失败(如系数矩阵 A 列线性相关)，还输出信息“fail”。

+1：运算成功，正常返回。

6. 函数说明

本函数要调用一般 $m\times n$ 阶的实矩阵 $\boldsymbol{QR}$ 分解函数 MatrixQR()，详细介绍请参阅第 5 章 5.6.2 节。

7. 示例

求下列超定方程组的最小二乘解，并求系数矩阵的 $\boldsymbol{QR}$ 分解式。

$$\begin{cases} x_0 + x_1 - x_2 = 2 \\ 2x_0 + x_1 = -3 \\ x_0 - x_1 = 1 \\ -x_0 + 2x_1 + x_2 = 4 \end{cases}$$

示例程序如下：

```
//LE_LinearLeastSquareHouseholder.cpp
//求解线性最小二乘问题的豪斯荷尔德变换法

#include <iostream>                    //输入输出流头文件
#include "LinearEquation.h"            //线性方程(组)求解头文件

void main()
{
  int i;
    double a[4][3] =
  {
      { 1.0,  1.0,  -1.0},
      { 2.0,  1.0,  0.0},
      { 1.0,  -1.0,  0.0},
      {-1.0,  2.0,  1.0}
  };

  double b[4] = {2.0, -3.0, 1.0, 4.0};

  matrix<double> qq(4, 4);
  valarray<double> bb(b, 4);
  matrix<double> aa(&a[0][0], 4, 3);

  i = LE_LinearLeastSquareHouseholder(aa, bb, qq);

  if(i != 0)
  {
      for(int s = 0; s < 3; s++)
      {
          cout << "x(" << s << ") = " << bb[s] << endl;
      }
      cout << endl;

      cout << "MAT Q IS: " << endl;
      MatrixLinePrint(qq);
      cout << endl;

      cout << "MAT R IS: " << endl;
```

```
        MatrixLinePrint(aa);
    }

    cout << endl;
}
```

8. 示例结果

```
x(0) =  -1.19048
x(1) =  0.952381
x(2) =  -0.666667

MAT Q IS:
    -0.377964    -0.377964     0.755929     0.377964
    -0.755929    -0.377964    -0.377964    -0.377964
    -0.377964     0.377964    -0.377964     0.755929
     0.377964    -0.755929    -0.377964     0.377964

MAT R IS:
    -2.64575    -2.22045e-016     0.755929
           0         -2.64575    -0.377964
           0                0    -1.13389
           0                0           0
```

9. 示例说明

示例运算所得结果中有的0元素不是严格的数值0，这是电子数字计算机进行数值计算本身性质所决定的。

7.2.12 线性最小二乘问题的广义逆法

1. 功能

利用广义逆求超定方程组 $\boldsymbol{AX}=\boldsymbol{B}$ 的最小二乘解。其中 $\boldsymbol{A}$ 为 $m \times n(m \geqslant n)$ 的矩阵，且列线性无关。当 $m = n$ 时，即为求线性方程组的解。

2. 算法

首先对矩阵 $\boldsymbol{A}$ 进行奇异值分解，即(参看第5章5.5.4节的算法说明)

$$\boldsymbol{A}=\boldsymbol{U}\begin{bmatrix}\sum & 0\\ 0 & 0\end{bmatrix}\boldsymbol{V}^{\mathrm{T}}$$

然后利用奇异值分解式计算 $\boldsymbol{A}$ 的广义逆 $\boldsymbol{A}^{+}$，即(参看第5章5.5.5节的算法说明)

$$\boldsymbol{A}^{+}=\boldsymbol{V}_1\sum{}^{-1}\boldsymbol{U}_1^{\mathrm{T}}$$

最后利用广义逆 $\boldsymbol{A}^{+}$ 求超定方程组 $\boldsymbol{AX}=\boldsymbol{B}$ 的最小二乘解，即 $\boldsymbol{X}=\boldsymbol{A}^{+}\boldsymbol{B}$。

3. 函数原型

```
//线性最小二乘问题的广义逆法
template <class _Ty>
int LE _LinearLeastSquareGeneralizedInverse(matrix<_Ty>& a,
        valarray<_Ty>& b, matrix<_Ty>& x, matrix<_Ty>& aa, _Ty eps,
```

matrix<_Ty>& u, matrix<_Ty>& v);

4. 参数

a:matrix<_Ty>类型引用,体积为 $m \times n$ 。存放超定方程组的系数矩阵 $\boldsymbol{A}$;返回时其对角线依次给出奇异值 $\sigma_0, \sigma_1, \cdots, \sigma_p$,其余元素均为零。

b:valarray<_Ty>类型引用,长度为 m。存放超定方程组右端常数向量。

x:valarray<_Ty>类型引用,长度为 n。返回时存放超定方程组的最小二乘解。

aa:matrix<_Ty>类型引用,体积为 $n \times m$ 。返回时存放系数矩阵 $\boldsymbol{A}$ 的广义逆 $\boldsymbol{A}^+$。

eps:_Ty 类型。奇异值分解函数中的控制精度要求。

u:matrix<_Ty>类型引用,体积为 $m \times m$ 。返回时存放系数矩阵 $\boldsymbol{A}$ 的奇异值分解式中的左奇异向量 U。

v:matrix<_Ty>类型引用,体积为 $n \times n$。返回时存放系数矩阵 $\boldsymbol{A}$ 的奇异值分解式中的右奇异向量 $\boldsymbol{V}^{\mathrm{T}}$。

5. 返回值

int 类型。

<0:表示在奇异值分解过程中迭代超过了 60 次还未达到精度要求,此时矩阵 $\boldsymbol{A}$ 的分解式为 $\boldsymbol{UAV}$,求最小二乘解失败。

>0:运算成功,正常返回。

6. 函数说明

本函数要调用求广义逆的奇异值分解的函数 GeneralizedInversionSingularValue(),详细介绍请参阅第 5 章的 5.5.5 节;在函数 GeneralizedInversionSingularValue()中又要调用一般实矩阵的奇异值分解的函数 MatrixSingularValue(),请参阅第 5 章的 5.5.4 节。

7. 示例

求下列超定方程组的最小二乘解,同时输出系数矩阵的广义逆:

$$\begin{cases} x_0 + x_1 - x_2 = 2 \\ 2x_0 + x_1 = -3 \\ x_0 - x_1 = 1 \\ -x_0 + 2x_1 + x_2 = 4 \end{cases}$$

取 $\varepsilon = 10^{-6}$ 。

示例程序如下:

```
//LE_LinearLeastSquareGeneralizedInverse.cpp  求解线性最小二乘问题的广义逆法

#include <iostream>                  //输入输出流头文件
#include "LinearEquation.h"          //线性方程(组)求解头文件

void main()
{
  int i;

  double a[4][3] =
  {
      { 1.0, 1.0, -1.0},
```

```
        { 2.0, 1.0, 0.0},
        { 1.0, -1.0, 0.0},
        {-1.0, 2.0, 1.0}
    };

    matrix<double> ta(&a[0][0], 4, 3);

    double b[4] = {2.0, -3.0, 1.0, 4.0};

     valarray<double> bb(b, 4);
     valarray<double> xx(3);
    matrix<double> taa(3, 4);
    matrix<double> uu(4, 4);
    matrix<double> vv(3, 3);

     double eps = FLOATERROR;

    i = LE_LinearLeastSquareGeneralizedInverse(ta, bb, xx, taa, eps, uu, vv);

    if (i>0)
    {
       for(i = 0; i < 3; i ++)
       {
           cout << "x(" << i << ") = " << xx[i] << endl;
       }
       cout << endl;

       cout << "MAT A+ : " << endl;
       MatrixLinePrint(taa);
       cout << endl;
   }
}
```

8. 示例结果

```
x(0) = -1.190476
x(1) = 0.952381
x(2) = -0.666667

MAT A+ :
    -0.047619      0.380952      0.238095     -0.047619
     0.238095      0.095238     -0.190476      0.238095
    -0.666667      0.333333      0.333333      0.333333
```

7.2.13 病态方程组求解

1. 功能

求解病态线性方程组 $\boldsymbol{AX}=\boldsymbol{B}$。

2. 算法

设线性方程组 $\boldsymbol{AX}=\boldsymbol{B}$ 是病态的。其求解的步骤如下。

(1) 用全选主元高斯消元法求解，得到一组近似解 $\boldsymbol{X}^{(1)}=(x_0^{(1)},x_1^{(2)},\cdots,x_{n-1}^{(1)})^{\mathrm{T}}$。

(2) 计算剩余向量

$$\boldsymbol{R}=\boldsymbol{B}-\boldsymbol{AX}^{(1)}$$

(3) 用全选主元高斯消元法求解线性方程组

$$\boldsymbol{AE}=\boldsymbol{R}$$

(4) 计算

$$\boldsymbol{X}^{(2)}=\boldsymbol{X}^{(1)}+\boldsymbol{E}$$

(5) 令 $\boldsymbol{X}^{(1)}=\boldsymbol{X}^{(2)}$，转到(2)，重复这个过程，直到

$$\max_{0\leqslant i\leqslant n-1}\frac{|x_i^{(2)}-x_i^{(1)}|}{1+|x_i^{(2)}|}<\epsilon$$

为止。

3. 函数原型

```
//病态方程组求解
template <class _Ty>
int LE_IllConditionedEquation(matrix<_Ty>& a, valarray<_Ty>& b,_Ty eps,
    matrix<_Ty>& x);
```

4. 参数

a:matrix<_Ty>类型引用，体积为 $n\times n$。存放方程组的系数矩阵。

b:valarray<_Ty>类型引用，长度为 n。存放方程组右端的常数向量。

eps:_Ty 类型。控制精度要求。

x:valarray<_Ty>类型引用，长度为 n。返回方程组的解向量。

5. 返回值

int 类型。

0:表示求解失败(如系数矩阵奇异)。

1:运算成功，正常返回。

6. 函数说明

本函数要调用全选主元高斯消元法求解线性方程组的函数 LE_TotalChoiceGauss()。详细介绍请参阅第 6 章 6.2.1 节。

本函数还调用了矩阵乘法函数 MatrixMultiply()，该函数请参阅第 5 章 5.2.3 节。

7. 示例

求下列 4 阶病态方程组

$$\begin{bmatrix}3.4336 & -0.5238 & 0.67105 & -0.15272\\ -0.5238 & 3.28326 & -0.73051 & -0.2689\\ 0.67105 & -0.73051 & 4.02612 & 0.01835\\ -0.15272 & -0.2689 & 0.01835 & 2.75702\end{bmatrix}\begin{bmatrix}x_0\\x_1\\x_2\\x_3\end{bmatrix}=\begin{bmatrix}-1.0\\1.5\\2.5\\-2.0\end{bmatrix}$$

取 $\varepsilon = 10^{-6}$ 。

示例程序如下：

```
//LE_IllConditionedEquation.cpp  病态方程组的求解

#include <iostream>                    //输入输出流头文件
#inclnde "LinearEquation.h"            //线性方程(组)求解头文件

void main()
{
  int k;
   double eps, x[4] = {0.0};
   double a[4][4] =
{
     { 3.4336, -0.5238, 0.67105, -0.15272},
     {-0.5238, 3.28326, -0.73051, -0.2689},
     { 0.67105, -0.73051, 4.02612, 0.01835},
     {-0.15272, -0.2689, 0.01835, 2.75702}
  };

  double b[4] = { -1.0, 1.5, 2.5, -2.0};

  valarray<double> bb(b, 4);
  valarray<double> xx(x, 4);
  matrix<double> aa(&a[0][0], 4, 4);

  eps = FLOATERROR;

  if(LE_IllConditionedEquation(aa, bb, eps, xx) > 0)
  {
      for(k = 0; k < 4; k++)
      {
          cout << "x[" << k << "] = " << xx[k] << endl;
      }
  }

  cout << endl;
}
```

8. 示例结果

```
x[0] = -0.397718
x[1] = 0.510054
x[2] = 0.782984
x[3] = -0.702916
```

第8章　非线性方程(组)求解

本章介绍非线性方程和非线性方程组的根(解)的各种数值算法与程序，每种算法都用C++语言以函数的形式实现。8.1节介绍非线性方程和非线性方程组的根(解)的基本概念，以及求解非线性方程和非线性方程组根(解)的各种算法程序原型的头文件。8.2节给出了求解非线性方程和非线性方程组根(解)的各种算法程序的使用示例及相关说明。

8.1　非线性方程和非线性方程组与头文件

本节介绍非线性方程和非线性方程组的根(解)的基本概念和相关头文件，其各种算法的函数程序原型包含在头文件 NonLinearEquation.h 中，函数的定义内容放在 NonLinearEquation.inl 文件中。

本章介绍的求解非线性方程和非线性方程组根(解)的算法及程序包括：

- 用二分法搜索方程 $f(x)=0$ 在区间$[a,b]$内的全部实根；
- 用牛顿法求解非线性方程的一个实根；
- 用埃特金法求解非线性方程的一个实根；
- 用连分式法求解非线性方程的一个实根；
- 用 *QR* 法求代数方程的全部根；
- 用牛顿下山法求解实系数代数方程的全部根(实根和复根)；
- 用牛顿下山法求解复系数代数方程的全部根(实根和复根)；
- 用梯度法(最速下降)求解非线性方程组的一组实根；
- 用拟牛顿法求解非线性方程组的一组实根；
- 非线性方程组最小二乘解的广义逆法；
- 用蒙特卡洛法求解非线性方程 $f(x)=0$ 的一个实根；
- 用蒙特卡洛法求解实(复)函数方程 $f(x)=0$ 的一个复根；
- 用蒙特卡洛法求解非线性方程组 $F(x)=0$ 的一组实根。

8.1.1　基本概念

1. 方程的根

设 $f(x)$为非线性函数，对于方程 $f(x)=0$，存在常数 c，使方程 $f(c)=0$，则称 c 为方程 $f(x)=0$ 的根。

2. 方程组的根(解)

设 $F(x)$为非线性方程组，对于方程组 $F(x)=0$，存在常向量 C，使方程组 $F(C)=0$，则称 C 为方程组 $F(x)=0$ 的根(解)。其中

$$F(x)=\begin{bmatrix} f_1(x) \\ f_2(x) \\ \cdots \\ f_n(x) \end{bmatrix} \qquad C=\begin{bmatrix} c_1 \\ c_2 \\ \cdots \\ c_n \end{bmatrix}$$

n 为正整数，$n>0$。

3. 多项式的根

设 $f(x)$ 为多项式 $p(x)$，c 为一常数，若有 $p(c)=0$，则称 c 为多项式 $p(x)$ 的零点或代数方程 $p(x)=0$ 的根。

8.1.2 NonLinearEquation.h

```
//NonLinearEquation.h 非线性方程(组)求解函数(方法)声明

#ifndef _NONLINEAREQUATION_H
#define _NONLINEAREQUATION_H

#include <comm.h>                    //公共头文件
#include <math.h>                    //数学头文件
#include <random.h>                  //随机数头文件
#include <matrix.h>                  //模板类 matrix 标准头文件
#include <LinearEquation.h>          //线性方程(组)求解头文件
#include <EigenvalueVector.h>        //计算特征值特征向量头文件

//用二分法搜索方程 f(x) = 0 在区间[a,b]内的全部实根
template <class _Ty>
inline size_t
RootHalves(_Ty a, _Ty b, _Ty step, _Ty eps, valarray<_Ty>& x, size_t m);

//用牛顿(Newton)法求解非线性方程的一个实根
template <class _Ty>
inline int
RootNewton( _Ty& x, _Ty eps, size_t js);

//用埃特金(Aitken)法求解非线性方程的一个实根
template <class _Ty>
inline int
RootAitken(_Ty& x, _Ty eps, size_t js);

//用连分式(Fraction)法求解非线性方程的一个实根
template <class _Ty>
inline int
RootFraction(_Ty& x, _Ty eps);

//用 QR 法求代数方程的全部根
```

```
template <class _Ty, class _Tz>
inline int
RootQR(valarray<_Ty>& a, valarray<_Tz>& x, _Ty eps, size_t jt);

//用牛顿下山(NewtonHillDown)法求解实系数代数方程的全部根(实根和复根)
template <class _Ty, class _Tz>
inline int
RootNewtonHillDown(valarray<_Ty>& a, valarray<_Tz>& cx);

//用牛顿下山(NewtonHillDown)法求解复系数代数方程的全部根(实根和复根)
//重载 RootNewtonHillDown()
template <class _Ty>
inline int
RootNewtonHillDown(complex<_Ty> a[], valarray< complex<_Ty> >& cx);

//用梯度(Gradient)法(最速下降)求解非线性方程组的一组实根
template <class _Ty>
inline int
RootGradient(_Ty eps, valarray<_Ty>& x, size_t js);

//用拟牛顿(QuasiNewton)法求解非线性方程组的一组实根
template <class _Ty>
inline int
RootQuasiNewton(_Ty eps, _Ty t, _Ty h, valarray<_Ty>& x, int k);

//非线性方程组最小二乘解的广义逆法
template <class _Ty>
int RootLeastSquareGeneralizedInverse(int m, _Ty eps1, _Ty eps2,
                                      valarray<_Ty>& x, int ka);

//用蒙特卡洛(MonteCarlo)法求解非线性方程 f(x) = 0 的一个实根
//f(x)的自变量为与系数都为实数
template <class _Ty>
inline void
RootMonteCarloComplex(_Ty& x, _Ty b, int m, _Ty eps);

//用蒙特卡洛(MonteCarlo)法求解实(复)函数方程 f(x) = 0 的一个复根
template <class _Tz, class _Ty>
inline void
RootMonteCarloComplex(_Tz& cxy, _Ty b, int m, _Ty eps);

//用蒙特卡洛(MonteCarlo)法求解非线性方程组 F(x) = 0 的一组实根
//f(x)的自变量为与系数都为实数
template <class _Ty>
```

```
inline void
RootMonteCarloGroupReal(valarray<_Ty>& x, _Ty b, int m, _Ty eps);

#include <NonLinearEquation.inl>

#endif //_NONLINEAREQUATION_H
```

8.2 各种非线性方程(组)计算方法

8.2.1 二分法

1. 功能

用二分法搜索方程 $f(x)=0$ 在区间[0,1]内的实根。

2. 算法

从端点 $x_0 = a$ 开始，以 h 为步长，逐步往后进行搜索。

对于第一个子区间$[x_i, x_{i+1}]$(其中 $x_{i+1} = x_i + h$)：

若 $f(x_i) = 0$，则 x_i 为一个实根，且从 $x_i + \frac{h}{2}$ 开始再往后搜索；若 $f(x_{i+1}) = 0$，则 x_{i+1}为一个实根，且从 $x_{i+1} + \frac{h}{2}$ 开始再往后搜索；若 $f(x_i)f(x_{i+1}) > 0$，则说明当前子区间内无实根，从 x_{i+1}开始再往后搜索；若 $f(x_i)f(x_{i+1}) < 0$，则说明在当前子区间内有实根。此时，反复将子区间减半，直到发现一个实根或子区间长度小于 ε 为止。在后一种情况下，子区间的中点即取为方程的一个实根。然后再从 x_{i+1}开始往后搜索。其中 ε 为预先给定的精度要求。以上过程一直进行到区间右端点 b 为止。

在使用本方法时，要注意步长 h 的选择。若步长 h 选得过大，可能会导致某些实根的丢失；若步长 h 选得过小，则会增加计算工作量。

3. 函数原型

```
//用二分法搜索方程 f(x) = 0 在区间[a,b]内的全部实根
template <class _Ty>
inline size_t
RootHalves(_Ty a, _Ty b, _Ty step, _Ty eps, valarray<_Ty>& x, size_t m);
```

4. 参数

a：_Ty 类型变量。求根区间的左端点。

b：_Ty 类型变量。求根区间的右端点。

h：_Ty 类型变量。搜索求根时采用的步长。

eps：_Ty 类型变量。控制精度要求。

x：valarray<_Ty>类型引用，长度为 m。返回区间$[a,b]$内搜索到的实根，其实根个数由函数值返回。

m：size_t 类型变量，在$[a,b]$内有实根个数的预估值。

5. 返回值

size_t 类型。

返回函数在区间[a,b]内实际搜索到的实根个数。若该值等于 m 时，则有可能在求根区间[a,b]内的实根未搜索完。

6. 函数说明

(1) 本函数需要调用计算方程左端函数 $f(x)$值的函数 FunctionValueRH()，需要由用户编写，其形式为：

```
template <class _Ty>
_Ty FunctionValueRH(_Ty dx)
{
   _Ty y;
   ......
   y = f(x)的表达式;
   return (y);
}
```

(2) 所有输入和输出的数据都应使用浮点类型。

7. 示例

求方程

$$f(x) = x^6 - 5x^5 + 3x^4 + x^3 - 7x^2 + 7x - 20 = 0$$

在区间[-2,5]内的所有实根。取步长 $h = 0.2$， $\varepsilon = 0.000001$。

由于本方程为 6 次代数方程，最多有 6 个实根，因此取 $m = 6$。

主函数程序与计算 $f(x)$的函数程序如下：

```
//RootHalves.cpp                                  二分法求解非线性方程示例

#include <iostream>                               //输入输出流头文件
#include "polynomials.h"                          //多项式及连分式求值头文件
#include "NonLinearEquation.h"                    //非线性方程(组)求解头文件
using namespace std;                              //名字空间

void main(void)
{
    size_t m = 6;                                 //估计实根个数
    size_t rootNo;                                //返回搜索到的实根个数
  valarray<long double> x(6);                     //搜索到的实根

  long double a = -2.0, b = 5.0, step = 0.2, eps = FLOATERROR;

    rootNo = RootHalves(a, b, step, eps, x, m);   //求根

  cout << "The amount of root is : " << rootNo << endl << endl;

    for(size_t i = 0; i < rootNo; i++)
      cout << "x(" << i << ") = " << x[i] << endl;
}
```

```
//*  计算 f(x)函数值
template <class _Ty>
_Ty FunctionValueRH(_Ty dx)
{
  _Ty dCoff[7] = {-20,7,-7,1,3,-5,1}, dValue;

  valarray<_Ty> vdCoff(dCoff, 7);                    //多项式系数

  dValue = PolyValueOneDim(vdCoff, 7, dx);           //计算多项式值

  return(dValue);
}
```

8. 示例结果

```
The amount of root is : 2
x(0) =  -1.40246
x(1) = 4.33376
```

9. 示例说明

示例中用到计算方程左端函数 $f(x)$值的函数 FunctionValueRH()，用户可根据具体情况灵活编写。

由于示例中要计算的非线性函数为 $f(x)$多项式，故在计算其函数值时调用了 PolyValueOneDim()函数，关于该函数请参阅第 4 章 4.2.1 节。

8.2.2 实根牛顿法

1. 功能

用牛顿(Newton)法求方程 $f(x)=0$ 的一个实根。

2. 算法

设方程 $f(x)=0$ 满足以下条件：

(1) $f(x)$在闭区间$[a,b]$上，其 $f'(x)$与 $f''(x)$均存在，且各自保持固定符号；

(2) $f(a)f(b)<0$；

(3) $f(x_0)f''(x)>0$，且 $x,x_0\in[a,b]$。

方程 $f(x)=0$ 在区间$[a,b]$上有且只有一个实根，取初值 x_0，由牛顿迭代格式

$$x_{n+1}=x_n-f(x_n)/f'(x_n)$$

计算得到的序列 $x_0,x_1,\cdots,x_n,\cdots$收敛于方程 $f(x)=0$ 的根。

结束迭代过程的条件为 ($|f(x_{n+1})|<\varepsilon$) 与 ($|x_{n+1}-x_n|<\varepsilon$) 同时成立，其中 ε 为预先给定的精度要求。

3. 函数原型

```
//用牛顿(Newton)法求解非线性方程的一个实根
template <class _Ty>
inline int
RootNewton( _Ty& x, _Ty eps, size_t js);
```

4. 参数

x：_Ty 类型变量引用。单元中存放迭代初值；返回时存放迭代终值。

eps：_Ty 类型变量。控制精度的要求。

js：整型变量。最大迭代次数。

5. 返回值

int 类型。

<0：表示牛顿迭代公式中出现了 $f'(x)=0$ 的情况。

>=js：表示迭代 js 次还未满足精度要求，返回的实根只能作为参考。

>=0 且<js：表示正常返回。

6. 函数说明

(1) 本函数需要调用计算 $f(x)$函数值及 $f'(x)$导数值的函数 FunctionValueRN()，由用户自己编写，其形式为：

```
template <class _Ty>
void FunctionValueRN(_Ty dx, valarray<_Ty>& y)
{
    ......
    y[0] = f(x)的表达式;
    y[1] = f'(x)的表达式;
}
```

(2) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用牛顿迭代法求方程 $f(x)=x^3-x^2-1=0$ 在 $x_0=1.5$ 附近的一个实数根。取 $\epsilon=0.000001$，最大迭代次数为 60，其中 $f'(x)=3x^2-2x$。

主函数及计算 $f(x)$函数值和 $f'(x)$导数值的程序如下：

```
//RootNewton.cpp                    用 Newton 法求解非线性方程的一个实根

#include <iostream>                 //输入输出流头文件
#include "polynomials.h"            //多项式及连分式求值头文件
#include "NonLinearEquation.h"      //非线性方程(组)求解头文件
using namespace std;                //名字空间

void main(void)
{
  int js = 60, k;
    long double eps = 0.000001, x = 1.5;
  cout << "RootNewton()" << endl << endl;
    k = RootNewton(x,eps,js);   //求根
    if(k >= 0)
      cout << "k = " << k << " x = " << x << endl;
}

//*   计算函数及导数值
template <class _Ty>
void FunctionValueRN(_Ty dx, valarray<_Ty>& y)
```

```
{
  _Ty dCoff0[4] = {-1, 0, -1, 1};
  _Ty dCoff1[3]= {0, -2, 3};
  valarray<_Ty> vdCoff0(dCoff0,4), vdCoff1(dCoff1,3);
  y[0] = PolyValueOneDim(vdCoff0, 4, dx);  //计算多项式值
  y[1] = PolyValueOneDim(vdCoff1, 3, dx);  //计算多项式导数值
}
```

8. 示例结果

```
RootNewton()
k = 4   x = 1.46557
```

9. 示例说明

示例中用到计算 $f(x)$函数值及 $f'(x)$导数值的函数 FunctionValueRN(),用户可根据具体情况灵活编写。

由于示例中要计算的非线性函数 $f(x)$为多项式,故在计算其函数值和导数值时调用了 PolyValueOneDim()函数,关于该函数请参阅第 4 章 4.2.1 节。

8.2.3 实根埃特金迭代法

1. 功能

用埃特金(Aitken)法求非线性方程 $x = \varphi(x)$ 的一个实数根。

2. 算法

设非线性方程为

$$x = \varphi(x)$$

取初值 x_0。埃特金迭代一次的过程如下：

预报　　$u = \varphi(x_k)$

再预报　　$v = \varphi(u)$

校正　　$x_{k+1} = v - \dfrac{(v-u)^2}{v-2u+x_k}$

结束迭代过程的条件为

$$|v-u| < \varepsilon$$

此时,v 即为非线性方程的一个实根。其中 ε 为预先给定的精度要求。

3. 函数原型

```
//用埃特金(Aitken)法求解非线性方程的一个实根
template <class _Ty>
inline int
RootAitken(_Ty& x, _Ty eps, size_t js);
```

4. 参数

x：　_Ty 类型变量引用。单元中存放迭代初值;返回时存放迭代终值。

eps：_Ty 类型变量。控制精度的要求。

js：　整型变量。最大迭代次数。

5. 返回值

int 类型。

<0:表示牛顿迭代公式中出现了 $f'(x)=0$ 的情况。

>=js:表示实际迭代次数已等于最大迭代次数,还不能满足精度要求,此时返回的实根值只作为参考。

>=0 且<js:表示正常返回。

6. 函数说明

本函数需要调用计算 $\varphi(x)$ 值的函数 FunctionValueRA(),由用户自己编写,其形式为:

```
template <class _Ty>
_Ty FunctionValueRA(_Ty dx)
{
   _Ty y;
   ......
   y = φ(x)的表达式;
   return (y);
}
```

7. 示例

用埃特金迭代法求方程 $x=6-x^2$ 在 $x_0=0.0$ 附近的一个实数根。取 $\varepsilon=0.000001$,最大迭代次数为 20。

主函数及计算 $\varphi(x)$ 值的函数程序如下:

```
//RootAitken.cpp                    用 Aitken 法求解非线性方程的一个实根

#include <iostream>                 //输入输出流头文件
#include "polynomials.h"            //多项式及连分式求值头文件
#include "NonLinearEquation.h"      //非线性方程(组)求解头文件
using namespace std;                //名字空间

void main(void)
{
  int js(20), k;
  long double eps = 0.0000001, x = 0;
  cout << "RootNewton()" << endl << endl;
   k = RootAitken(x, eps, js);
   if (k >= 0)
     cout << "k = " << k << "\t x = " << x << endl;
}

//*计算函数值
template <class _Ty>
_Ty FunctionValueRA(_Ty dx)
{
  _Ty dCoff[3] = {6, 0, -1}, dValue;
  valarray<_Ty> vdCoff(dCoff, 3);
  dValue = PolyValueOneDim(vdCoff, 3, dx);  //计算多项式值
  return(dValue);
```

```
}
```

8. 示例结果

```
RootNewton()
k=12  x = 2
```

9. 示例说明

示例中用到计算 $\varphi(x)$ 函数值的函数 FunctionValueRA()，用户可根据具体情况灵活编写。

由于示例中要计算的函数 $\varphi(x)$ 为多项式，故在计算其函数值和导数值时调用了 PolyValueOneDim()函数，关于该函数请参阅第 4 章 4.2.1 节。

8.2.4 实根连分式法

1. 功能

使用连分式求非线性方程 $f(x)=0$ 的一个实数根。

2. 算法

设非线性方程为 $f(x)=0$，而 $f(x)$ 的反函数为 $x=F(y)$，并用连分式表示为

$$x=a_0+\frac{y-y_0}{a_1}+\frac{y-y_1}{a_2}+\cdots+\frac{y-y_{i-1}}{a_i}+\cdots$$

其中 $y_i=f(x_i)$。因此，满足方程 $f(x)=0$ 的根为

$$x=a_0-\frac{y_0}{a_1}-\frac{y_1}{a_2}-\cdots-\frac{y_{i-1}}{a_i}-\cdots$$

综上所述，可以得到求非线性方程 $f(x)=0$ 的一个实根的过程如下。

选取初值 x_0 及 x_1，并计算出 $y_0=f(x_0)$，$y_1=f(x_1)$。根据 (x_0,y_0) 及 (x_1,y_1) 可以计算出

$$a_0=x_0$$
$$a_1=(y_1-y_0)/(x_1-x_0)$$

由此可以计算出 $x_2=a_0-\frac{y_0}{a_1}$。

一般来说，若已知点列 $(x_0,y_0),(x_1,y_1),\cdots,(x_{i-1},y_{i-1})$，并已求得 $a_0,a_1,\cdots,a_{i-1}$，则可以计算出

$$x_i=a_0-\frac{y_0}{a_1}-\frac{y_1}{a_2}-\cdots-\frac{y_{i-2}}{a_{i-1}}$$

及

$$y_i=f(x_i)$$

然后通过下列递推公式计算出 a_i。

$$a_0=x_0$$
$$a_{0i}=x_i$$
$$a_{j+1,i}=\frac{y_i-y_j}{a_{ji}-a_j},j=0,1,\cdots i-1$$
$$a_i=a_{ii}$$

以上过程一直进行到 $|y_i|<\varepsilon$ 为止。其中 ε 为预先给定的精度要求。

在实际计算过程中，上述的连分式最多作到七节为止。如果此时还不满足精度要求，则将最后得到的实根近似值作为新的初值再重新计算。

本方法对函数 $f(x)$ 的要求比较低，在有实根的情况下，一般都能通过本方法求出一个实根。

3. 函数原型

```
//用连分式(Fraction)法求解非线性方程的一个实根
template <class _Ty>
inline int
RootFraction(_Ty& x, _Ty eps);
```

4. 参数

x：_Ty 类型变量引用。单元中存放迭代初值；返回时存放迭代终值。

eps：_Ty 类型变量。控制精度的要求。

5. 返回值

int 类型。

=10：表示迭代了 10 次，可能没有满足精度要求，返回的实根只作为参考。

<10：表示正常返回。

6. 函数说明

(1) 本函数需要调用计算 $f(x)$ 函数值的函数 FunctionValueRF()，需要由用户自己编写，其形式为：

```
template <class _Ty>
_Ty FunctionValueRH(_Ty dx)
{
   _Ty y;
   ......
   y = f(x)的表达式;
   return (y);
}
```

(2) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用连分式法求方程 $f(x) = x^3 - x^2 - 1 = 0$ 在 $x_0 = 1.0$ 附近的一个实数根。取 $\varepsilon = 0.000001$。

主函数程序及计算 $f(x)$ 函数值的程序如下：

```
//RootFraction.cpp                    用 Fraction 法求解非线性方程的一个实根

#include <iostream>                   //输入输出流头文件
#include "polynomials.h"              //多项式及连分式求值头文件
#include "NonLinearEquation.h"        //非线性方程(组)求解头文件
using namespace std;                  //名字空间
void main(void)
{
    double eps = DOUBLEERROR, x = 1.0;
  cout << "RootFraction()" << endl << endl;
```

```
    int k = RootFraction(x, eps);
    if(k >= 0)
      cout << "k = " << k << " x = " << x << endl;
  cout << endl;
}

//*计算函数值
template <class _Ty>
_Ty FunctionValueRF(_Ty dx)
{
  _Ty dCoff[4] = {-1, 0, -1, 1}, dValue;
  valarray<_Ty> vdCoff(dCoff, 4);
  dValue = PolyValueOneDim(vdCoff, 4, dx);        //计算多项式值
  return(dValue);
}
```

8. 示例结果

```
RootFraction()
k = 1 x = 1.46557
```

9. 示例说明

示例中用到计算 $f(x)$ 函数值的函数 FunctionValueRF()，用户可根据具体情况灵活编写。

由于示例中要计算的非线性函数 $f(x)$ 为多项式，故在计算其函数值时调用了 PolyValueOneDim()函数，关于该函数请参阅第 4 章 4.2.1 节。

8.2.5 实系数代数方程全部根的 *QR* 法

1. 功能

用 *QR* 法求实系数 n 次多项式方程

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = 0$$

的全部根。

2. 算法

令

$$b_i = a_i / a_n, i = n-1, \cdots, 1, 0$$

则一般实系数 n 次多项式方程 $P_n(x) = 0$ 化为 n 次首一多项式方程

$$Q_n(x) = x^n + b_{n-1}x^{n-1} + \cdots b_1 x + b_0 = 0$$

由线性代数可知，$Q_n(x) = 0$ 可以看成是如下实矩阵

$$Q = \begin{bmatrix} -b_{n-1} & -b_{n-2} & -b_{n-3} & \cdots & -b_1 & -b_0 \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

的特征多项式方程

$$Q_n(\lambda) = \lambda^n + b_{n-1}\lambda^{n-1} + \cdots + b_1\lambda + b_0 = 0$$

因此，求方程 $Q_n(x) = 0$ 的全部根就变为求上述 Q 矩阵的全部特征值。矩阵 Q 为一个上 H 阵，可以直接用 QR 方法求出全部特征值。

有关 QR 方法求上 H 阵全部的问题请参阅第 6 章 6.2.4 节。

3. 函数原型

```
//QR法求代数方程全部根
template <class _Ty, class _Tz>
inline int
RootQR(valarray<_Ty>& a, valarray<_Tz>& x, _Ty eps, size_t jt);
```

4. 参数

a：valarray<_Ty>类型引用，长度为 $n+1$，存放 n 次多项式方程

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = 0$$

的 $n+1$ 个系数。

x：valarray<_Tz>类型引用，长度为 n，返回 n 个根。

eps：_Ty 类型变量。QR 方法中的控制精度要求。

jt：非负整型变量。控制 QR 方法中最大迭代次数。

5. 返回值

int 类型。

<0：表示在 QR 方法中迭代已经超过最大迭代次数，但还未满足精度要求。

>0：表示正常返回。

6. 函数说明

本函数在计算过程中，需要调用 QR 方法求上 H 阵全部特征值的函数 EigenvalueVectorHessenbergQR()，有关内容请参阅第 6 章 6.2.4 节。

所有输入数据都应为浮点型。特别地，由于所求到的根不一定全是实根，故函数的第二个参数设为 valarray<complex<_Ty> >类型，用于存放复数根。

7. 示例

用 QR 方法求 6 次实系数多项式方程

$$P_6(x) = 1.5x^6 - 7.5x^5 + 4.5x^4 + 1.5x^3 - 10.5x^2 + 10.5x - 30 = 0$$

的全部根。取 $\varepsilon = 0.000001$，最大迭代次数为 60。

主函数程序如下：

```
//RootQR.cpp                          用QR法求实系数代数方程的全部根

#include <iostream>                   //输入输出流头文件
#include "NonLinearEquation.h"        //非线性方程(组)求解头文件
using namespace std;                  //名字空间

void main(void)
{
  size_t i, jt = 60, n = 6;
    double eps = DOUBLEERROR;
```

```
    valarray<complex<double> > cx(6);

    double a[7] = {-30.0, 10.5, -10.5, 1.5, 4.5, -7.5, 1.5};
  valarray<double> va(a, 7);

  cout << "RootQR()" << endl << endl;

  i = RootQR(va, cx, eps, jt);          //求根

    if (i > 0)
      for (i=0; i<6; i++)
          cout << "cx(" << i << ") = " << cx[i] << endl;

  cout << endl;
}
```

8. 示例结果

```
RootQR()

cx(0) = (-1.40246,0)
cx(1) = (4.33376,0)
cx(2) = (1.18398,-0.936099)
cx(3) = (1.18398,0.936099)
cx(4) = (-0.149622,-1.19251)
cx(5) = (-0.149622,1.19251)
```

9. 示例说明

从示例中可以看到，所求到的根只有头两个为实根(其根的虚数部分为0)，故RootQR()函数的第二个参数设为valarray<complex<_Ty> >类型，其中含有所求的全部实根与复根。

8.2.6 代数方程全部根的牛顿—下山法

1. 功能

用牛顿—下山法求代数方程

$$f(z) = a_n z^n + a_{n-1} z^{n-1} + \cdots + a_1 z + a_0 = 0$$

的全部根。

代数方程的系数可是实数，也可是复数。

2. 算法

牛顿-下山法的迭代格式为

$$z_{i+1} = z_i - t f(z_i)/f'(z_i)$$

选取适当的 t 可以保证有

$$|f(z_{i+1})|^2 < |f(z_i)|^2$$

迭代过程一直作到 $|f(z_i)|^2 < \varepsilon$ 为止。

迭代格式在鞍点或接近重根点时，可能因 $f'(z) \approx 0$ 而 $|f(z)|^2 \neq 0$ 而失败。在本函数中

采用撒网络的方法，即选取适当的 d 与 c，用

$$x_{i+1} = x_i + d\cos(c)$$

$$y_{i+1} = y_i + d\sin(c)$$

计算，使 $|f(z_{i+1})|^2 < |f(z_i)|^2$ 而冲过鞍点或使

$$|f(z)|^2 < \varepsilon$$

而求得一个根。

每当求得一个根 z^* 后，在 $f(z)$ 中劈去因子 $(z-z^*)$，再求另一个根。

以上过程直到求出全部根为止。

在实际计算时，每一个根都要作变换 $z = \sqrt[n-1]{|a_0|}z'$，使得当 $a_n = 1$ 时，$|a_0| = 1$，保证寻根在单位圆内进行。

3. 函数原型

```
//用牛顿下山(NewtonHillDown)法求解实系数代数方程的全部根
template <class _Ty, class _Tz>
inline int
RootNewtonHillDown(valarray<_Ty>& a, valarray<_Tz>& cx);

//用牛顿下山(NewtonHillDown)法求解复系数代数方程的全部根
template <class _Ty>
inline int
RootNewtonHillDown(complex<_Ty> a[], valarray< complex<_Ty> >& cx);
```

4. 参数

对于第一个函数：

a：valarray<_Tz>类型引用。代数方程实系数。

cx：valarray<_Tz>类型引用。返回时存放方程的解。

对于第二个函数：

a：complex<_Tz>类型数组。代数方程复系数。

cx：valarray<complex<_Tz> >类型引用。返回时存放方程的解。

5. 返回值

int 类型。

<0：表示方程中的所有系数 $a_j = 0(j=0,1,2,\ldots,n)$。

>0：表示正常返回。

6. 函数说明

(1) 对于应用牛顿·下山法求代数方程，定义了两个函数，其他函数名及参数的个数都相同，区别仅在于参数的类型不一样。第一个函数为实系数代数所用，其第一个参数为实系数而准备；第二个函数为复系数代数所用，其第一个参数为复系数而准备。两个方程的第二参数实际作用是一样的，在函数返回时存放有代数方程的解。在调用任何一个函数时，只要正确地赋予参数，程序会自动正确地调用相应的函数。

(2) 所有输入和输出的数据都应使用浮点类型。

7. 示例

(1) 用牛顿—下山法求解实系数代数方程

$$f(z) = z^6 - 5z^5 + 3z^4 + z^3 + 7z^2 - 20 = 0$$

的全部根。

(2) 用牛顿—下山法求解复系数代数方程

$$f(z) = z^5 + (3+3j)z^4 - 0.01jz^3 + (4.9 - 19j)z^2 + 21.33z + (0.1 - 100j) = 0$$

的全部根。

主函数程序如下：

```
//RootNewtonHillDown.cpp
//用牛顿下山法求解代数方程的全部根(实根和复根)

#include <iostream>                    //输入输出流头文件
#include "NonLinearEquation.h"         //非线性方程(组)求解头文件
using namespace std;                   //名字空间

void main(void)
{
  int i, ii;

  cout << "RootNewtonHillDown()" << endl << endl;
//验证实系数
  valarray< complex<double> > ca(6);
  double a[7] = {-20, 7, -7, 1, 3, -5, 1};
  valarray<double> va(a, 7);

  cout << "Result (Algebra Equation with Real Coefficient): " << endl << endl;

    i = RootNewtonHillDown(va, ca);
    if (i > 0)
      for(ii = 0; ii < 6; ii++)
          cout << "ca(" << ii << ") = " << ca[ii] << endl;
//验证复系数
  complex<double> ccb[6] =
  {
      complex<double>(0.1, -100.0), complex<double>(21.33, 0.0),
      complex<double>(4.9, -19.0), complex<double>(0.0, -0.01),
      complex<double>(3.0, 2.0), complex<double>(1.0, 0.0)
  };

  valarray< complex<double> > chb(5);

  cout << endl << "Result (Algebra Equation with Complex Coefficient): " << endl << endl;

  i = RootNewtonHillDown(ccb, cbb);
    if (ii > 0)
```

```
    for(size_t ii = 0; ii < 5; ii++)
        cout << "cb(" << ii << ") = " << cbb[ii]<< endl;
}
```

8. 示例结果

```
RootNewtonHillDown()

Result (Algebra Equation with Real Coefficient):

ca(0) = (4.33376,0)
ca(1) = (-0.149622,1.19251)
ca(2) = (-0.149622,-1.19251)
ca(3) = (-1.40246,0)
ca(4) = (1.18398,0.936099)
ca(5) = (1.18398,-0.936099)

Result (Algebra Equation with Complex Coefficient):

cb(0) = (-2.16668,-2.7672)
cb(1) = (2.09386,1.10221)
cb(2) = (-3.43095,0.304085)
cb(3) = (0.543454,-2.18746)
cb(4) = (-0.0396854,1.54836)
```

9. 示例说明

示例中分别对一个实系数代数方程和一个复系数代数文件进行了求解，前者的解中有实数解，也有复数解，后者的解全为复数。但调用的求解函数名都是相同的，实参个数也相同，只是实参的类型有所不同。

8.2.7 非线性方程组一组实根的梯度法

1. 功能

用梯度法(即最速下降法)求函数方程组

$$f_i(x_0,x_1,\cdots,x_{n-1})=0,\quad i=0,1,\cdots,n-1$$

的一组实根。

2. 算法

设非线性方程组为

$$f_i=f_i(x_0,x_1,\cdots,x_{n-1})=0,\quad i=0,1,\cdots,n-1$$

并定义目标函数为

$$F=F(x_0,x_1,\cdots,x_{n-1})=\sum_{i=0}^{n-1}f_i^2$$

则梯度的计算过程如下。

(1) 选取一组初值 $x_0,x_1,\cdots,x_{n-1}$ 。

(2) 计算目标函数值

$$F = F(x_0, x_1, \cdots, x_{n-1}) = \sum_{i=0}^{n-1} f_i^2$$

(3) 若 $F < \varepsilon$，则 $X = (x_0, x_1, \cdots, x_{n-1})^T$ 即为方程的一组实根，过程结束；否则继续。

(4) 计算目标函数在 $(x_0, x_1, \cdots, x_{n-1})$ 的偏导数

$$\frac{\partial F}{\partial x_i} = 2\sum_{j=0}^{n-1} f_j \cdot \frac{\partial f_j}{\partial x_i}, \quad i = 0, 1, \cdots, n-1$$

再计算

$$D = \sum_{j=0}^{n-1} \left(\frac{\partial F}{\partial x_j}\right)^2$$

(5) 计算

$$x_i - \lambda \frac{\partial F}{\partial x_i} \Rightarrow x_i, \quad i = 0, 1, \cdots, n-1$$

其中 $\lambda = F/D$。

重复步骤(2)～(5)，直到满足精度要求为止。

在上述过程中，如果 $D = 0$，则说明遇到了目标函数的局部极值点，此时可改变初值再试一试。

3. 函数原型

```
//用梯度(Gradient)法(最速下降)求解非线性方程组的一组实根
template <class _Ty>
inline int
RootGradient(_Ty eps, valarray<_Ty>& x, size_t js);
```

4. 参数

Eps：_Ty 类型变量。控制精度要求。

x：valarray<_Ty>类型引用，长度为 n。存放一组初值 $x_0, x_1, \cdots, x_{n-1}$；返回时存放方程组的一组实根。

js：非负整型变量。允许最大迭代次数。

5. 返回值

int 类型（本函数返回实际迭代次数）。

<0：说明在迭代过程中遇到了目标函数的局部极值点，即

$$D = \sum_{j=0}^{n-1} \left(\frac{\partial F}{\partial x_j}\right)^2 = 0$$

此时可改变初值重新再试。

=js：说明迭代了 js 次还未满足精度要求。此时可改变下列条件之一再进行尝试。

- 改变初值；
- 放松精度要求；
- 增大 js。

6. 函数说明

(1) 本函数需调用计算目标函数值及偏导数值的函数 FunctionValueRG()，由用户自己编写，其形式为：

```
template <class _Ty>
_Ty FunctionValueRG(.....)
```

```
{
    _Ty z;
    ......
    z = Σ_{i=0}^{n-1} f_i^2 的表达式;
    y[0] = 2Σ_{j=0}^{n-1} f_j · ∂f_j/∂x_0 的表达式;
          ⋮
    y[n-1] = 2Σ_{j=0}^{n-1} f_j · ∂f_j/∂x_{n-1} 的表达式;
    return(z);
}
```

(2) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用梯度法求下列非线性方程组的一组实根：

$$\begin{cases} f_0 = x_0 - 5x_1^2 + 7x_2^2 + 12 = 0 \\ f_1 = 3x_0x_1 + x_0x_2 - 11x_0 = 0 \\ f_2 = 2x_1x_2 + 40x_0 = 0 \end{cases}$$

取初值为(1.5,6.5,-5.0),精度要求 ε 为 0.000001,最大迭代次数为 500。

主函数程序及计算目标函数值、偏导数值的函数程序如下：

```
//RootGradient.cpp                    用 Gradient 法求解非线性方程组的一组实根

#include <iostream>                   //输入输出流头文件
#include "polynomials.h"              //多项式及连分式求值头文件
#include "NonLinearEquation.h"        //非线性方程(组)求解头文件
using namespace std;                  //名字空间

void main(void)
{
  int js = 500, i;
    double eps = FLOATERROR, x[3] = {1.5, 6.5, -5.0};
  valarray<double> vx(x, 3);

    i = RootGradient(eps, vx, js);    //求根

  cout << "RootGradient()" << endl << endl;

    if ((i>0)&&(i<js))
      for(i = 0; i < 3; i++)
          cout << "x(" << i << ") = " << vx[i] << endl;

  cout << endl;
}
```

```
//*   计算函数及导数值
template <class _Ty>
_Ty FunctionValueRG(valarray<_Ty>& x, valarray<_Ty>& y)
{
  _Ty z,f1,f2,f3,df1,df2,df3;

    f1 = x[0] - 5.0 * x[1] * x[1] + 7.0 * x[2] * x[2] + 12.0;     //计算二维多项式 f0 函数值
    f2 = 3.0 * x[0] * x[1] + x[0] * x[2] - 11.0 * x[0];           //计算二维多项式 f1 函数值
    f3 = 2.0 * x[1] * x[2] + 40.0 * x[0];                         //计算二维多项式 f2 函数值
    z = f1 * f1 + f2 * f2 + f3 * f3;
    df1 = 1.0;                              //计算二维多项式 f0 关于 x0 偏导数值
  df2 = 3.0 * x[1] + x[2] - 11.0;           //计算二维多项式 f1 关于 x0 偏导数值
  df3 = 40.0;                               //计算二维多项式 f2 关于 x0 偏导数值
    y[0] = 2.0 * (f1 * df1 + f2 * df2 + f3 * df3);
    df1 = 10.0 * x[1];
    df2 = 3.0 * x[0];
  df3 = 2.0 * x[2];
    y[1] = 2.0 * (f1 * df1 + f2 * df2 + f3 * df3);
    df1 = 14.0 * x[2];
  df2 = x[0];
  df3 = 2.0 * x[1];
    y[2] = 2.0 * (f1 * df1 + f2 * df2 + f3 * df3);
    return(z);
}
```

8. 示例结果

```
RootGradient()

x(0) =  1.00019
x(1) =  5.00043
x(2) =  -4.00039
```

9. 示例说明

示例中调用了计算目标函数值、偏导数值的函数程序，可以由用户自己编写，也可以由专门的程序自动获得。

8.2.8 非线性方程组一组实根的拟牛顿法

1. 功能

用拟牛顿法求非线性方程组

$$f_i(x_0,x_1,\cdots,x_{n-1})=0,\quad i=0,1,\cdots,n-1$$

的一组实数解。

2. 算法

设非线性方程组

$$f_i(x)=0,\quad i=0,1,\cdots,n-1$$

其中　　$X=(x_0,x_1,\cdots,x_{n-1})^{\mathrm{T}}$

若假设 X 的第 k 次迭代近似值为

$$X^{(k)}=(x_0^{(k)},x_1^{(k)},\cdots,x_{n-1}^{(k)})^{\mathrm{T}}$$

则计算第 $k+1$ 次迭代值的牛顿迭代格式为

$$X^{(k+1)}=X^{(k)}-F(X^{(k)})^{-1}f(X^{(k)})$$

其中

$$f(X^{(k)})=(f_0^{(k)},f_1^{(k)},\cdots,f_{n-1}^{(k)})^{\mathrm{T}}$$

$$f_i^{(k)}=f_i(X^{(k)})$$

$F(X)$为雅可比(Jacobi)矩阵，即

$$F(X)=\begin{bmatrix}\dfrac{\partial f_0(X)}{\partial x_0} & \dfrac{\partial f_0(X)}{\partial x_1} & \cdots & \dfrac{\partial f_0(X)}{\partial x_{n-1}}\\ \dfrac{\partial f_1(X)}{\partial x_0} & \dfrac{\partial f_1(X)}{\partial x_1} & \cdots & \dfrac{\partial f_1(X)}{\partial x_{n-1}}\\ \vdots & \vdots & & \vdots\\ \dfrac{\partial f_{n-1}(X)}{\partial x_0} & \dfrac{\partial f_{n-1}(X)}{\partial x_1} & \cdots & \dfrac{\partial f_{n-1}(X)}{\partial x_{n-1}}\end{bmatrix}$$

令　　$\delta^{(k)}=F(X^{(k)})^{-1}f(X^{(k)})$

其中　　$\delta^{(k)}=(\delta_0^{(k)},\delta_1^{(k)},\cdots,\delta_{n-1}^{(k)})^{\mathrm{T}}$

则有　　$F(X^{(k)})\delta^{(k)}=f(X^{(k)})$

$$X^{(k+1)}=X^{(k)}-\delta^{(k)}$$

若在雅可比矩阵中，用差商代替偏导数，

即

$$\frac{\partial f_i(X^{(k)})}{\partial x_j}\approx\frac{f_i(X_j^{(k)})-f_i(X^{(k)})}{h}$$

其中 h 足够小，且

$$f_i(X_j^{(k)})=f_i(x_0^{(k)},\cdots,x_{j-1}^{(k)},x_j^{(k)}+h,x_{j+1}^{(k)},\cdots,x_{n-1}^{(k)})$$

则方程组(2)变为

$$\sum_{j=0}^{n-1}f_i(X_j^{(k)})z_j^{(k)}=f_j(X^{(k)}),\quad i=0,1,\cdots,n-1$$

其中

$$z_j^{(k)}=\frac{\delta_j^{(k)}}{h+\sum_{s=0}^{n-1}\delta_s^{(k)}},\quad j=0,1,\cdots,n-1$$

综上所述，拟牛顿法解非线性方程组的计算过程如下。

取初值 $X=(x_0,x_1,\cdots,x_{n-1})^{\mathrm{T}}$，$h>0$，$0<t<1$。

(1) 计算 $f_i(X)\Rightarrow B(i)$，　$i=0,1,\cdots,n-1$。

(2) 若满足

$$\max_{0\leqslant i\leqslant n-1}|B(i)|<\varepsilon$$

则方程组的一组实数解为

$$X=(x_0,x_1,\cdots,x_{n-1})^{\mathrm{T}}$$

计算过程结束；否则继续。

(3) 计算

$$f_i(X_j)\Rightarrow A(i,j),i,j=0,1,\cdots,n-1$$

其中 $X=(x_0,x_1,\cdots,x_{j-1},x_j+h,x_{j+1},\cdots,x_{n-1})^{\mathrm{T}}$。

(4) 解线性代数方程组 $AZ=B$,其中 $Z=(z_0,z_1,\cdots,z_{n-1})^{\mathrm{T}}$,且计算 $\beta=1-\sum_{j=0}^{n-1}z_j$。

(5) 计算 $x_i-hz_i/\beta\Rightarrow x_i,i=0,1,\cdots,n-1$。

(6) $t*h\Rightarrow h$,转(1)。

以上过程一直作到 X 满足精度要求为止。

在计算过程中,若发现线性方程组 $AZ=B$ 奇异或 $\beta=0$(即 $\sum_{j=0}^{n-1}z_j=1$)的情况,可以采取以下一些措施试一试:

- 放宽精度要求 ε;
- 适当改变 t 与 h 的初值;
- 改变 X 的初值;
- 改变方程组中各方程的顺序。

3. 函数原型

```
//用拟牛顿(QuasiNewton)法求解非线性方程组的一组实根
template <class _Ty>
inline int
RootQuasiNewton(_Ty eps, _Ty t, _Ty h, valarray<_Ty>& x, int k);
```

4. 参数

eps:_Ty 类型变量。控制精度要求。

t:_Ty 类型变量。控制 h 大小的变量,$0<t<1$。

h:_Ty 类型变量。增量初值,在本函数中要被破坏。

x:valarray<_Ty>类型引用,长度为 n。存放初值 $(x_0^{(0)},x_1^{(0)},\cdots,x_{n-1}^{(0)})$,返回时存放方程组的一组实数解 $(x_0,x_1,\cdots,x_{n-1})$。

k:整型变量。最多允许的迭代次数。

5. 返回值

int 类型 。

0:说明迭代了 k 次还未满足精度要求,程序工作失败,并输出信息"fail!"。

-1:说明线性方程组 $AZ=B$ 奇异,并输出信息"Fail!"。

-2:说明 $\beta=0$,即 $\sum_{j=0}^{n-1}z_j=1$,并输出信息"Fail!"。

>0:运行正常,返回实际迭代次数。

6. 函数说明

(1) 在本函数的计算过程中要调用全选主元高斯消元法求线性代数方程组的函数 LE_TotalChoiceGauss(),参阅第 7 章 7.2.1 节。

(2) 在本函数中还要调用计算方程组左端函数值 $f_i(X)$ 的函数 FunctionValueRLN(),由用户自己编写,其形式为

```
template <class _Ty>
void FunctionValueRLN(valarray<_Ty>& x, valarray<_Ty>& y)
{
    ……
```

$y[0]=f_0(x_0,x_1,\cdots,x_{n-1})$的表达式；

⋮

$y[n-1]=f_{n-1}(x_0,x_1,\cdots,x_{n-1})$的表达式；

}

(3) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用拟顿法求非线性方程组

$$\begin{cases} f_0 = x_0^3 + x_1^2 + x_2^2 - 1 = 0 \\ f_1 = 2x_0^2 + x_1^2 - 4x_2 = 0 \\ f_2 = 3x_0^2 - 4x_1 + x_2^2 = 0 \end{cases}$$

的一组实数解。取初值为 $(1.0,1.0,1.0)^T$，$t=0.1$，$h=0.1$，$eps=0.0000001$，最大迭代次数 $k=100$。

主函数程序及计算左端函数值 $f_i(X)$ 的函数程序如下：

```
//RootQuasiNewton.cpp          用拟牛顿法求解非线性方程组的一组实根

#include <iostream>                                    //输入输出流头文件
#include "polynomials.h"                               //多项式及连分式求值头文件
#include "NonLinearEquation.h"                         //非线性方程(组)求解头文件
using namespace std;                                   //名字空间

void main(void)
{
  int k = 100, i;
   double eps = 0.000001, x[3] = {1.0, 1.0, 1.0}, t = 0.1, h = 0.1;
   valarray<double> vx(x, 3);
  cout << "RootQuasiNewton()" << endl << endl;
   i = RootQuasiNewton(eps, t, h, vx, k);              //求根
  cout << "i = " << i << endl << endl;
  for(i = 0; i < 3; i++)
      cout << "x(" << i << ") = " << vx[i] << endl;
  cout << endl;
}

//* 计算函数值
template <class _Ty>
void FunctionValueRSN(valarray<_Ty>& x, valarray<_Ty>& y)
{
  y[0] = x[0] * x[0] + x[1] * x[1] + x[2] * x[2] - 1.0;      //计算多项式 f0 函数值
  y[1] = 2.0 * x[0] * x[0] + x[1] * x[1] - 4.0 * x[2];       //计算多项式 f1 函数值
  y[2] = 3.0 * x[0] * x[0] - 4.0 * x[1] + x[2] * x[2];       //计算多项式 f2 函数值
}
```

8. 示例结果

```
RootQuasiNewton()
```

```
i = 4

x(0) = 0.785197
x(1) = 0.496611
x(2) = 0.369923
```

9. 示例说明

示例中调用了计算目标函数组函数值的程序，该程序可以由用户自己编写，也可以由专门的程序自动获得。

8.2.9 非线性方程组最小二乘解的广义逆法

1. 功能

选用广义逆求解无约束条件下的优化问题

$$f_i(x_0,x_1,\cdots,x_{n-1})=0,i=0,1,\cdots,m-1;m\geqslant n$$

当 $m=n$ 时，即为求解非线程组。

2. 算法

设非线性方程组为

$$f_i(x_0,x_1,\cdots,x_{n-1})=0,i=0,1,\cdots,m-1;m\geqslant n$$

其雅可比(Jacobi)矩阵为

$$\boldsymbol{A}=\begin{bmatrix}\dfrac{\partial f_0}{\partial x_0} & \dfrac{\partial f_0}{\partial x_1} & \cdots & \dfrac{\partial f_0}{\partial x_{n-1}}\\ \dfrac{\partial f_1}{\partial x_0} & \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_{n-1}}\\ \vdots & \vdots & & \vdots\\ \dfrac{\partial f_{m-1}}{\partial x_0} & \dfrac{\partial f_{m-1}}{\partial x_1} & \cdots & \dfrac{\partial f_{m-1}}{\partial x_{n-1}}\end{bmatrix}$$

计算非线性方程最小二乘解的迭代公式为

$$X^{(k+1)}=X^{(k)}-\alpha_k Z^{(k)}$$

其中，$Z^{(k)}$ 为线性方程组 $\boldsymbol{A}^{(k)}Z^{(k)}=F^{(k)}$ 的线性最小二乘解，即

$$Z^{(k)}=(A^{(k)})+F^k$$

式中，$\boldsymbol{A}^{(k)}$ 为 k 次迭代值 $X^{(k)}$ 的雅可比矩阵；$F^{(k)}$ 为 k 次迭代值的左端函数值，即

$$F^{(k)}=(f_0^{(k)},f_1^{(k)},\cdots,f_{m-1}^{(k)})^{\mathrm{T}}$$

$$f_i^{(k)}=f_i(x_0^{(k)},x_1^{(k)},\cdots,x_{n-1}^{(k)}),i=0,1,\cdots,m-1$$

a_k 为使 a 的一元函数 $\sum_{i=0}^{m-1}(f_i^{(k+1)})^2$ 达到极小值的点。在本函数中利用有理极值法计算 a_k。

3. 函数原型

```
//非线性方程组最小二乘解的广义逆法
template <class _Ty>
int RootLeastSquareGeneralizedInverse(int m, _Ty eps1, _Ty eps2,
                                      valarray<_Ty>& x, int ka);
```

4. 参数

m：整形变量。非线性方程组中方程个数。

eps1：_Ty 类型变量。控制最小二乘解的精度要求。

eps2：_Ty 类型变量。用于奇异值分解中的控制精度要求。

x：valarray<_Ty>类型引用，长度为 n。存放非线性方程组解的初始近似值 $X^{(0)}$，要求各分量 $x_0^{(0)},x_1^{(0)},\cdots,x_{n-1}^{(0)}$ 不全为零；返回时存放最小二乘解，当 $m=n$ 时，即为非线性方程组的解。

ka：整型变量。$ka=\max\{m,n\}+1$。

5. 返回值

int 类型。

<0：说明在奇异值分解中迭代超过了 60 次还未达到精度要求。

=0：说明本函数迭代了 60 次还未达到精度要求。

>0：表示正常返回。

6. 函数说明

(1) 本函数在计算过程中要调用以下函数：

① 利用广义逆求解线性最小二乘问题的 LE_LinearLeastSquareGeneralizedInverse()函数，请参阅第 7 章 7.2.12 节；

② 在函数 LE_LinearLeastSquareGeneralizedInverse()中要调用求解广义逆的函数 GencralizedInversionSingularValue()，请参阅第 5 章 5.6.4 节；

③ 在函数 GeneralizedInversionSingularValue()中要调用求解广义逆的函数 MatrixSingularValue()，请参阅第 5 章 5.6.3 节。

(2) 计算非线性方程组中各方程左端函数 $f_i(x_0,x_1,\cdots,x_{n-1})(i=0,1,\cdots,m-1)$ 的函数，由用户自己编写，其形式为：

```
//计算非线性方程组左边函数值
template <class _Ty>
void FunctionValueRLSGI(valarray<_Ty>& x, valarray<_Ty>& d)
{
   d[0] = f0(x0,x1,…,xn-1)的表达式;
     ⋮
   d[m-1] = fn-1(x0,x1,…,xn-1)的表达式;
}
```

(3) 计算雅可比矩阵函数，由用户自己编写，其形式为：

```
//计算雅可比矩阵
template <class _Ty>
void JacobiMatrix(valarray<_Ty>& x, matrix<_Ty>& p)
{
    p(i,j) = ∂fi/∂xj (i=0,1,…,m-1;j=0,1, …,n-1)的表达式;
}
```

注：(2)与(3)中的 m 与 n 值分别为方程的个数和方程中未知数的个数。

7. 示例

(1) 求解非线性方程组

$$\begin{cases} x_0^2+10x_0x_1+4x_1^2+0.7401006=0 \\ x_0^2-3x_0x_1+2x_1^2-1.0201228=0 \end{cases}$$

其中 $m=2,n=2$。取初值为 $(0.5,-1.0)$,$eps1=0.000001$,$eps2=0.000001$。

(2) 求非线性方程组

$$\begin{cases} x_0^2+7x_0x_1+3x_1^2+0.5=0 \\ x_0^2-2x_0x_1+x_1^2-1=0 \\ x_0+x_1+1=0 \end{cases}$$

的最小二乘解。其中 $m=3,n=2$。取初值 $(1.0,-1.0)$,$eps1=eps2=0.000001$

主函数程序及计算非线性方程组左端函数值的函数程序,计算雅可比矩阵的函数的示例程序有两个,分别如下。

示例一:

```
//RootLeastSquareGeneralizedInverse1.cpp
//非线性方程组最小二乘解的广义逆法示例 1

#include <iostream>          //输入输出流头文件
#include "LinearEquation.h"  //线性方程组求解头文件
#include "NonLinearEquation.h"//非线性方程(组)求解头文件
using namespace std;         //名字空间

void main(void)
{
  int m(2), i, ka(3);
   double eps1(FLOATERROR), eps2(FLOATERROR);
   valarray<double> x(2);
  x[0] = 0.5;
  x[1] = -1.0;
  cout << "RootLeastSquareGeneralizedInverse1()" << endl;
  i = RootLeastSquareGeneralizedInverse(m, eps1, eps2, x, ka);
   cout << endl;
  cout << "i = " << i << endl << endl;
   for (int it = 0; it < 2; it++)
    cout << "x(" << it << ") = " << x[it] << endl;
  cout<<endl;
}

//计算非线性方程组左边函数值
template <class _Ty>
void FunctionValueRLSGI(valarray<_Ty>& x,valarray<_Ty>& d)
{
 d[0] = x[0] * x[0] + 10.0 * x[0] * x[1] + 4.0 * x[1] * x[1] + 0.7401006;
  d[1] = x[0] * x[0] - 3.0 * x[0] * x[1] + 2.0 * x[1] * x[1] - 1.0201228;
}

//计算雅可比矩阵
template <class _Ty>
```

```
void JacobiMatrix(valarray<_Ty>& x, matrix<_Ty>& p)
{
   p(0,0) = 2.0 * x[0] + 10.0 * x[1];
    p(0,1) = 10.0 * x[0] + 8.0 * x[1];
    p(1,0) = 2.0 * x[0] - 3.0 * x[1];
    p(1,1) = -3.0 * x[0] + 4.0 * x[1];
}
```

示例二：

```
//RootLeastSquareGeneralizedInverse2.cpp
//非线性方程组最小二乘解的广义逆法示例 2

#include <iostream>                    //输入输出流头文件
#include "LinearEquation.h"            //线性方程组求解头文件
#include "NonLinearEquation.h"         //非线性方程(组)求解头文件
using namespace std;                   //名字空间

void main(void)
{
  int m(3), i, ka(4);
   double eps1(FLOATERROR), eps2(FLOATERROR);
   valarray<double> x(2);

  x[0] = 1.0;
  x[1] = -1.0;

  cout << "RootLeastSquareGeneralizedInverse2()" << endl;

  i = RootLeastSquareGeneralizedInverse(m, eps1, eps2, x, ka);

   cout << endl;
  cout << "i = " << i << endl << endl;

   for (i=0; i<2; i++)
     cout << "x(" << i << ") = " << x[i] << endl;

  cout<<endl;
}

//计算非线性方程组左边函数值
template <class _Ty>
void FunctionValueRLSGI(valarray<_Ty>& x,valarray<_Ty>& d)
{
    d[0] = x[0] * x[0] + 7.0 * x[0] * x[1] + 3.0 * x[1] * x[1] + 0.5;
    d[1] = x[0] * x[0] - 2.0 * x[0] * x[1] + x[1] * x[1] - 1.0;
```

```
    d[2] = x[0] + x[1] + 1.0;
}

//计算雅可比矩阵
template <class _Ty>
void JacobiMatrix(valarray<_Ty>& x, matrix<_Ty>& p)
{
   p(0,0) = 2.0 * x[0] + 7.0 * x[1];
   p(0,1) = 7.0 * x[0] + 6.0 * x[1];
   p(1,0) = 2.0 * x[0] - 2.0 * x[1];
   p(1,1) = -2.0 * x[0] + 2.0 * x[1];
   p(2,0) = 1.0;
   p(2,1) = 1.0;
}
```

8. 示例结果

结果一：

```
RootLeastSquareGeneralizedInverse1()

i = 1

x(0) = 0.376547
x(1) = -0.437953
```

结果二：

```
RootLeastSquareGeneralizedInverse2()

i = 1

x(0) = 0.378947
x(1) = -0.692576
```

8.2.10 非线性方程一个实根的蒙特卡洛法

1. 功能

用蒙特卡洛(MonteCarlo)法求非线性方程 $f(x)=0$ 的一个实数根。$f(x)$ 的自变量为与系数都为实数。

2. 算法

设函数方程为

$$f(x)=0$$

用蒙特卡洛法求一个实根的过程如下。

选一个初值 x，并计算 $F_0=f(x)$，再选一个 $b>0$。

在区间 $[-b,b]$ 上反复产生均匀分布的随机数 r，对于第一个 r 计算 $F_1=f(x+r)$，直到发现一个 r 使 $|F_1|<|F_0|$ 为止，此时 $x+r\Rightarrow x$，$F_1\Rightarrow F_0$。如果连续产生了 m 个随机数 r 还不满足 $|F_1|<|F_0|$，则将 b 减半再进行。

重复上述过程，直到$|F_0|<\varepsilon$为止，此时的x即为非线性方程$f(x)=0$的一个实根。

在使用本方法时，如遇到迭代不收敛，则可以适当调整b与m的值。

3. 函数原型

```
//用蒙特卡洛(MonteCarlo)法求解非线性方程 f(x) = 0 的一个实根
template <class _Ty>
inline void
RootMonteCarloComplex(_Ty& x, _Ty b, int m, _Ty eps);
```

4. 参数

x：_Ty 类型变量引用。单元存放初值；返回方程根的终值。

b：_Ty 类型变量。均匀分布随机数的端点初值。

m：整形变量。控制调节点 b 的参数。

eps：_Ty 类型变量。控制精度要求。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数需要调用产生 0 到 1 之间均匀分布的一个随机数的函数 rand_01_One()，请参阅第 2 章 2.2.2 节。

(2) 本函数还要调用计算方程左端数值 $f(x)$ 的函数 FunctionValueMCR()，由用户自己编写，其形式为：

```
template <class _Ty>
_Ty FunctionValueMCR(_Ty x)
{
    _Ty y;
    ......
    y = f(x)的表达式;
    return(y);
}
```

(3) 本函数中程序 $f(x)=0$ 左边函数的自变量为与系数都为实数。

7. 示例

用蒙特卡洛法求实函数方程

$$f(x)=e^{-x^3}-\frac{\sin x}{\cos x}+800=0$$

在区间 $\left(0,\frac{\pi}{2}\right)$ 内的一个实根。

取初值 $x=0.5, b=1.0, m=10, \varepsilon=0.00001$。

主函数程序及计算 $f(x)$ 的函数程序如下：

```
//RootMonteCarloReal.cpp          用 MonteCarlo 法求解非线性方程的一个实根

#include <iostream>               //输入输出流头文件
#include "NonLinearEquation.h"    //非线性方程(组)求解头文件
using namespace std;              //名字空间
```

```
void main(void)
{
  long double x(0.5);              //自变量
  int m(10);
  long double eps(DOUBLEERROR), b(1);

  cout << "RootMonteCarloReal()" << endl << endl;

  RootMonteCarloReal(x,b,m,eps);
  cout << "x = " << x << endl << endl;
}

//*  计算函数值
template <class _Ty>
_Ty FunctionValueMCR(_Ty x)        //f(x)的自变量为实数
{
  _Ty y;

  y = exp(-x*x*x) - sin(x) / cos(x) + 800.0;

  return(y);
}
```

8. 示例结果

```
RootMonteCarloReal()

x = 1.56955
```

8.2.11 非线性方程一个复根的蒙特卡洛法

1. 功能

用蒙特卡洛(MonteCarlo)法求实函数或复函数方程 $f(z)=0$ 的一个复数根。

2. 算法

设非线性方程为

$$f(z)=0$$

左端函数 $f(z)$ 的模函数 $\|f(z)\|$。

蒙特卡洛法求一个复根的过程如下。

选取一个初值 $z=x+jy$，其中 $j=\sqrt{-1}$，并计算 $F_0=\|f(z)\|$，再选取一个 $b>0$。

在区间 $[-b,b]$ 上反复产生均匀分布的随机数 r_x 与 r_y，对于每一对 (r_x,r_y) 计算

$$F_1=\|f(x+r_x+j(y+r_y))\|$$

直到发现一对 (r_x,r_y) 使 $F_1<F_0$ 为止，此时

$$x+r_x+j(y+r_y)\Rightarrow z,F_1\Rightarrow F_0$$

如果连续产生了 m 对随机数 (r_x,r_y) 还不满足 $F_1<F_0$，则将 b 减半再进行。

重复上述过程，直到 $F_0<\varepsilon$ 为止，此时的 z 即为 $f(z)=0$ 的一个复根。

在使用本方法时，如遇到迭代不收敛，则可以适当调整 b 与 m 的值。

本方法可用于求只包含两个未知量的非线性方程组

$$\begin{cases} f_1(x,y)=0 \\ f_2(x,y)=0 \end{cases}$$

的一组实数根。此时将 x 当作实部，y 当作虚部。

3. 函数原型

```
//用蒙特卡洛(MonteCarlo)法求解实(复)函数方程 f(x) = 0 的一个复根
//f(x)的自变量为复数，或自变量与系数都为复数(不能都为实数)
template <class _Tz, class _Ty>
inline void
RootMonteCarloComplex(_Tz& cxy, _Ty b, int m, _Ty eps);
```

4. 参数

x：_Ty 类型变量指针。在此指针指向的单元中存放迭代初值；返回时存放迭代终值。

eps：_Ty 类型变量。控制精度的要求。

js：整型变量。最大迭代次数。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数需要调用产生 0 到 1 之间均匀分布的一个随机数的函数 rand_01_One()，请参阅第 2 章 2.2.2 节。

(2) 本函数中所计算的 $f(x)$ 的自变量可以为复数，也可以 $f(x)$ 的自变量与系数都为复数(不能都为实数)。

(3) 本函数还要调用计算模函数 $\|f(z)\|$ 的函数 FunctionModule()，由用户自己编写，其形式为：

```
template <class _Tz>
double FunctionModule(_Tz cxy)
{
    _Tz z;
    ……
    z = ‖f(x+jy)‖ 的表达式
    return(z)
}
```

7. 示例

示例使用了两个代数方程 $f(x)=0$ 作为计算的方程，一个是实代数方程，即其中的 $f(x)$ 的系数是实的，但自变量是复数；另一个为复代数方程，即 $f(x)$ 的系数与自变量都是复数。

(1) 求实函数方程

$$f(z)=z^2-6z+13=0$$

的一个复根。取初值 $z=0.5+0.5j$，$b=1.0$，$m=10$，$\varepsilon=0.00001$。

(2) 求复函数方程

$$f(z)=z^2+(1+j)z-2+2j=0$$

的一个复根。取初值 $z=0.5+0.5j$，$b=1.0$，$m=10$，$\varepsilon=0.00001$。

主函数程序及计算 $\|f(z)\|$ 值的函数程序如下：

```
//RootMonteCarloComplex.cp          用 MonteCarlo 法求解非线性方程的一个复根

#define FM      //注解此行,将调用 FunctionModule2()

#ifdef FM
  #define FunctionModule FunctionModule1
#else
  #define FunctionModule FunctionModule2
#endif

#include <iostream>              //输入输出流头文件
#include "polynomials.h"         //多项式及连分式求值头文件
#include "NonLinearEquation.h"   //非线性方程(组)求解头文件
using namespace std;             //名字空间

void main(void)
{
  int m = 10;
   double eps = FLOATERROR, b = 1;
  complex<double> cxy(0.5, 0.5);//自变量
  cout << "RootMonteCarloComplex()" << endl << endl;
   RootMonteCarloComplex(cxy,b,m,eps);
  cout << "z = " << cxy << endl;
}

//*  计算函数模
template <class _Tz>
double FunctionModule1(_Tz cxy)  //f(x)的自变量为复数
{
  _Tz cmodule(cxy);
  double dCoff[3] = {13, -6, 1};
  valarray<double> vdCoff(dCoff,3);
  cmodule = PolyValueOneDim(vdCoff, 3, cxy);
  return(Abs(cmodule));
}

//*  计算函数模
template <class _Tz>
double FunctionModule2(_Tz cxy)  //f(x)的自变量与系数都为复数
{
  _Tz cmodule(cxy);
  complex<double> cdCoff[3] =
  {
```

```
        complex<double>(-1,2),
        complex<double>(1,1),
        complex<double>(1,0)
    };
    valarray<complex<double> > vcdCoff(cdCoff,3);
    cmodule = PolyValueOneDim(vcdCoff, 3, cxy);
    return(Abs(cmodule));
}
```

8. 示例结果

```
//当调用函数 FunctionModule1 时,运行结果为

RootMonteCarloComplex()

z = (3,2)

//当调用函数 FunctionModule2 时,运行结果为

RootMonteCarloComplex()

z = (-1.6838,0.133552)
```

9. 示例说明

(1) 示例中所选用的计算方程为代数方程,其左边函数为多项式,因此在计算多项式值时使用了求多项式值的函数 PolyValueOneDim(),该函数对多项式的系数与自变量为实数或复数都能够进行计算,有关细节请参阅第 4 章 4.2.1 节。

(2) 为了简便,针对两种不同的条件,示例中将两个不同方程合并到一个程序中进行计算。对于方程的左边函数,一个的系数为实数,自变量复数,另一个的自变量与系数都为复数。在计算中需要对方程左边函数求模,为了方便,求模计算过程分别进行。示例中采用了条件编译的方法分别计算:在对第一个方程左边函数求模计算时,示例不用进行改动,程序只对第一个方程左边函数求模,即只会调用 FunctionModule1();而当要对第二个方程左边函数进行求模计算时,可将预编译命令进行注解处理,如下:

```
//#define FM
```

这时程序将只调用第二个求模函数 FunctionModule2()进行计算,不会调用第一个求模函数。

(3) 示例计算的结果有两个,为了方便,都统一给出。

8.2.12 非线性方程组一组实根的蒙特卡洛法

1. 功能

用蒙特卡洛(Monte Carlo)法求非线性方程组

$$f_i(x_0,x_1,\cdots,x_{n-1})=0,i=0,1,\cdots,n-1$$

的一组实根。$f(x)$ 的自变量与系数都为实数。

2. 算法

设非线性方程组为

$$f_i(x_0,x_1,\cdots,x_{n-1})=0,i=0.1,\cdots,n-1$$

定义模函数为

$$\|F\|=\sqrt{\sum_{i=0}^{n-1}f_i^2}$$

用蒙特卡洛法求一组实根的过程如下。

选取初值 $X=(x_0,x_1,\cdots,x_{n-1})^T$，并计算模函数值 $F_0=\|F\|$。再选取一个 $b>0$。

在区间 $[-b,b]$ 上反复产生均匀公布的随机数 $(r_0,r_1,\cdots,r_{n-1})$，对于每一组 $(r_0,r_1,\cdots,r_{n-1})$ 计算 $(x_0+r_0,x_1+r_1,\cdots,x_{n-1}+r_{n-1})^T$ 的模函数值 F_1，直到发现一组使 $F_1<F_0$ 为止，此时令

$$x_i+r_i\Rightarrow x_i,i=0,1,\cdots,n-1$$
$$F_1\Rightarrow F_0$$

如果连续产生了 m 组随机数还不满足 $F_1<F_0$，则将 b 减半再进行。

重复上述过程，直到 $F_0<\varepsilon$ 为止，此时的 $X=(x_0,x_1,\cdots,x_{n-1})^T$ 即为非线性方程组的一组实根。

在使用本办法时，如遇到迭代不满足收敛，则可以适当调整 b 与 m 的值。

3. 函数原型

```
//用蒙特卡洛(MonteCarlo)法求解非线性方程组 F(x) = 0 的一组实根
template <class _Ty>
inline void
RootMonteCarloGroupReal(valarray<_Ty>& x, _Ty b, int m, _Ty eps);
```

4. 参数

x：valarray<_Ty>类型引用，长度为 n。存放一组实根初值；返回一组实根终值。

b：_Ty 类型变量。随机数的端点初值。

m：整型变量。控制调节 b 的参数。

eps：_Ty 类型变量。控制精度要求。

5. 返回值

本函数无返回值。

6. 函数说明

(1)本函数需要调用产生 0 到 1 之间均匀分布的一个随机数的函数 rand_01_One()，请参阅第 2 章 2.2.2 节。

(2) $f(x)$ 的自变量与系数都为实数。

(3) 本函数还要调用计算模函数 $\|f(x)\|$ 的函数 FunctionModule()，由用户自己编写，其形式为：

```
template <class _Tz>
_Ty FunctionModule(valarray<_Ty> x)
{
    _Ty f;
    ……
    f = √(Σ_{i=0}^{n-1} f_i^2) 的表达式;
```

```
    return(f);
}
```

7. 示例

求非线性方程组

$$\begin{cases} 3x_1 + x_2 + 2x_3^2 - 3 = 0 \\ -3x_1 + 5x_2^2 + 2x_1x_3 - 1 = 0 \\ 25x_1x_2 + 20x_3 + 12 = 0 \end{cases}$$

的一组实根。取初值 $X = (0,0,0)^T, b = 2.0, m = 10, \varepsilon = 0.00001$ 。

主函数及计算模函数值的函数程序如下：

```
//RootMonteCarloGroupReal.cpp          用 MonteCarlo 法求解非线性方程组的一组实根

#include <iostream>                    //输入输出流头文件
#include "NonLinearEquation.h"         //非线性方程(组)求解头文件
using namespace std;                   //名字空间

void main(void)
{
  size_t m(10), n(3), i;
  valarray<double> x(3);//自变量
  double eps(DOUBLEERROR), b(2);
  x[0] = x[1] = x[2] = 0;
  cout << "RootMonteCarloGroupReal()" << endl << endl;
   RootMonteCarloGroupReal(x,b,m,eps);
   for(i = 0; i < n; i++)
     cout << "x(" << i << ") = " << x[i] << endl;
}

//*  计算函数模
template <class _Ty>
_Ty FunctionModule(valarray<_Ty> x)
{
  _Ty f,f1,f2,f3;

   f1 = 3.0 * x[0] + x[1] + 2.0 * x[2] * x[2] - 3.0;
   f2 = -3.0 * x[0] + 5.0 * x[1] * x[1] + 2.0 * x[0] * x[2] - 1.0;
   f3 = 25.0 * x[0] * x[1] + 20.0 * x[2] + 12.0
   f = sqrt(f1 * f1 + f2 * f2 + f3 * f3);
   return(f);
}
```

8. 示例结果

```
RootMonteCarloGroupReal()

x(0) = 1.09637
```

```
x(1) =  -0.799768
x(2) =  0.496039
```

第9章 插　　值

本章介绍插值的算法，每种算法都用C++语言以函数的形式实现。9.1节介绍插值的基本概念，以及插值算法程序原型的头文件。9.2节给出了计算各种插值算法的使用示例及相关说明。

9.1 插值与头文件

本节介绍插值基本概念，以及插值各种算法和函数程序原型，其算法和函数程序原型包含在头文件 Interpolation. h 中，函数的定义内容在 Interpolation. inl 文件中。

本章介绍的计算插值算法及程序包括：

- 一元全区间不等距插值；
- 一元全区间等距插值；
- 一元三点不等距插值；
- 一元三点等距插值；
- 连分式不等距插值；
- 连分式等距插值；
- 埃尔米特不等距插值；
- 埃尔米特等距插值；
- 埃特金不等距逐步插值；
- 埃特金等距逐步插值；
- 光滑不等距插值；
- 光滑等距插值；
- 第一种边界条件的三次样条函数插值、微商与积分；
- 第二种边界条件的三次样条函数插值、微商与积分；
- 第三种边界条件的三次样条函数插值、微商与积分；
- 二元三点插值；
- 二元全区间插值。

9.1.1 基本概念

1. 插值

在科学计算中经常会碰到许多计算上比较复杂，或者直接写不出其表达式的“困难”函数。解决这个难题的方法之一就是使用这些“困难”函数在某些点上的函数值或者加上其导数值，在某种规定和标准之下，构造出一些相对简单、性质又比较好的另一类“容易”函数，这种作法称为“插值”。

插值的目的是为了使用这后一类“容易”函数来近似代替前一类“困难”函数，即可以使用

这些“容易”函数的函数值或导数值来近似代替“困难”函数的函数值或导数值。

2. 插值函数

插值函数就是上述内容中提到的“容易”函数，常用的插值函数有多项式和三角函数。使用多项式作为插值函数，则称该多项式为“插值多项式”。

本章仅介绍使用多项式进行插值。有关多项式的介绍请参阅第 4 章“多项式与连分式函数计算”。

3. 插值点

在构造插值函数时要用到的“困难”函数在某些点上的函数值，这些点就称为插值函数的“插值点”。

9.1.2 Interpolation.h

```
// Interpolation.h       插值头文件
// Ver 1.0.0.0
// 版权所有(C)  何渝(HE Yu) 2002
// 最后修改：2002.7.19

#ifndef _INTERPOLATION_H       //避免多次编译
#define _INTERPOLATION_H

#include <valarray>            //数组模板类标准头文件
#include <Matrix.h>            //矩阵类头文件
#include <comm.h>              //公共头文件
using namespace std;           //名字空间

//一元全区间不等距插值
template <class _Ty>
_Ty Interpolation1VariableNotIsometry(valarray<_Ty>& x,
                                      valarray<_Ty>& y, _Ty t);

//一元全区间等距插值
template <class _Ty>
_Ty Interpolation1VariableIsometry(_Ty x0, _Ty h, valarray<_Ty>& y, _Ty t);

//一元三点不等距插值
template <class _Ty>
_Ty Interpolation1Variable3PointsNotIsometry(valarray<_Ty>& x,
                                             valarray<_Ty>& y, _Ty t);

//一元三点等距插值
template <class _Ty>
_Ty Interpolation1Variable3PointsIsometry(_Ty x0,_Ty h,
                                          valarray<_Ty>& y, _Ty t);
```

```
//连分式不等距插值
template <class _Ty>
_Ty InterpolationFractionNotIsometry(valarray<_Ty >& x,
                                                    valarray<_Ty>& y, _Ty t);

//连分式等距插值
template <class _Ty>
_Ty InterpolationFractionIsometry(_Ty x0, _Ty h, valarray<_Ty>& y, _Ty t);

//埃尔米特不等距插值
template <class _Ty>
_Ty InterpolationHermiteNotIsometry(valarray<_Ty>& x,
                                    valarray<_Ty>& y, valarray<_Ty>& dy, _Ty t);

//埃尔米特等距插值
template <class _Ty>
_Ty InterpolationHermiteIsometry(_Ty x0, _Ty h,
                                valarray<_Ty>& y, valarray<_Ty>& dy, _Ty t);

//埃特金不等距逐步插值
template <class _Ty>
_Ty InterpolationAitkenNotIsometry(valarray<_Ty>& x,
                                        valarray<_Ty>& y, _Ty t, _Ty eps);

//埃特金等距逐步插值
template <class _Ty>
_Ty InterpolationAitkenIsometry(_Ty x, _Ty h,
                                        valarray<_Ty>& y, _Ty t, _Ty eps);

//光滑不等距插值
template <class _Ty>
void InterpolationSmoothNotIsometry(valarray<_Ty>& x,
                            valarray<_Ty>& y, int k, _Ty t, valarray<_Ty>& s);

//光滑等距插值
template <class _Ty>
void InterpolationSmoothIsometry(_Ty x, _Ty h,
                            valarray<_Ty>& y, int k, _Ty t, valarray<_Ty>& s);

//第一种边界条件的三次样条函数插值、微商与积分
template <class _Ty>
_Ty Interpolation3SpoolingIstBoundary(valarray<_Ty>& x,
                valarray<_Ty>& y, valarray<_Ty>& dy, valarray<_Ty>& ddy,
                    valarray<_Ty>& t, valarray<_Ty>& z, valarray<_Ty>& dz,
```

```
                                                    valarray<_Ty>& ddz);

//第二种边界条件的三次样条函数插值、微商与积分
template <class _Ty>
_Ty Interpolation3Spooling2ndBoundary(valarray<_Ty>& x,
                valarray<_Ty>& y, valarray<_Ty>& dy, valarray<_Ty>& ddy,
                    valarray<_Ty>& t, valarray<_Ty>& z, valarray<_Ty>& dz,
                                                    valarray<_Ty>& ddz);

//第三种边界条件的三次样条函数插值、微商与积分
template <class _Ty>
_Ty Interpolation3Spooling3thBoundary(valarray<_Ty>& x,
                valarray<_Ty>& y, valarray<_Ty>& dy, valarray<_Ty>& ddy,
                    valarray<_Ty>& t, valarray<_Ty>& z, valarray<_Ty>& dz,
                                                    valarray<_Ty>& ddz);

//二元三点插值
template <class _Ty>
_Ty Interpolation2Variable3Points(valarray<_Ty>& x, valarray<_Ty>& y,
                                                matrix<_Ty> z, _Ty u, _Ty v);

//二元全区间插值
template <class _Ty>
_Ty Interpolation2VariableWholeInterval(valarray<_Ty>& x,
                            valarray<_Ty>& y, matrix<_Ty> z, _Ty u, _Ty v);

#include "Interpolation.inl"     //类及相关函数的定义头文件
#endif       //_INTERPOLATION_H
```

9.2 各种插值计算方法

9.2.1 一元全区间不等距插值

1. 功能

给定 n 个不等距结点 $x_i(i=0,1,\cdots,n-1)$ 的函数值 $y_i=f(x_i)$，用拉格朗日(Lagrange)插值公式，计算指定插值点 t 处的函数近似值 $z=f(t)$。

2. 算法

为了避免龙格(Runge)现象对计算结果的影响，在 n 个结点中自动选择 8 个结点进行插值，且使指定插值点 t 位于它们的中间。即选取满足 $x_k<x_{k+1}<x_{k+2}<x_{k+3}<x_{k+4}<x_{k+5}<x_{k+6}<x_{k+7}$ 的 8 个结点，使用七次拉格朗日插值多项式计算插值点 t 处的函数近似值 $z=f(t)$，即

$$z=\sum_{i=k}^{k+7}y_i\prod_{\substack{j=k\\j\neq i}}^{k+7}[(t-x_j)/(x_i-x_j)]$$

当插值点 t 位于包含 n 个结点的区间时，将仅取区间某端的 4 个结点进行插值；而当插值点 t 靠近 n 个结点的两端时，选取的结点数将少于 8 个。

3. 函数原型

```
//一元全区间不等距插值
template <class _Ty>
_Ty Interpolation1VariableNotIsometry(valarray<_Ty>& x,
                                      valarray<_Ty>& y, _Ty t);
```

4. 参数

x：valarray<_Ty>类型引用，长度为 n。存放给定的 n 个结点的值 x_i，要求 $x_0 < x_1 < \cdots < x_{n-1}$。

y：valarray<_Ty>类型引用，长度为 n。存放 n 个给定结点上的函数值 $y_i = f(x_i)(i = 0,1,\cdots,n-1)$。

t：_Ty 类型变量。存放指定插值点的值。

5. 返回值

本函数返回一个_Ty 类型函数值，即在插值点 t 处的函数近似值 $x = f(t)$。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

已知一元不等距列表函数如下：

i	1	2	3	4	5
x_i	0.10	0.15	0.25	0.40	0.50
y_i	0.904837	0.860708	0.778801	0.670320	0.606531
i	6	7	8	9	10
x_i	0.57	0.70	0.85	0.93	1.00
y_i	0.565525	0.496585	0.427415	0.394554	0.367879

使用拉格朗日插值公式计算插值点 $t = 0.63$ 处的函数近似值。

示例程序如下：

```
//Interpolation1VariableNotIsometry.cpp     一元全区间不等距插值

#include <iostream.h>                 //输入输出流头文件
#include "Interpolation.h"            //插值头文件

void main()
{
    double t(0.63), s;
    double x[10] =
    {
        0.10, 0.15, 0.25, 0.40, 0.50,
        0.57, 0.70, 0.85, 0.93, 1.00
    };
```

```
    double y[10] =
    {
            0.904837, 0.860708, 0.778801, 0.670320, 0.606531,
            0.565525, 0.496585, 0.427415, 0.394554, 0.367879
    };
        valarray<double> yy(y,10);
        valarray<double> xx(x,10);
    cout << " Interpolation1VariableNotIsometry()" << endl << endl;
    s = Interpolation1VariableNotIsometry(xx,yy,t);
    cout<<" t = " << t << "\t s = "<< s << endl << endl;
}
```

8. 示例结果

```
Interpolation1VariableNotIsometry()

t = 0.63    s = 0.532591
```

9.2.2 一元全区间等距插值

1. 功能

给定 n 个等距结点 $x_i = x_0 + ih\,(i = 0,1,\cdots,n-1)$ 的函数值 $y_i = f(x_i)$，用拉格朗日(Lagrange)插值公式计算指定插值点 t 处的函数近似值 $z = f(t)$。

2. 算法

同本章 9.2.1 节。

3. 函数原型

```
//一元全区间等距插值
template <class _Ty>
_Ty Interpolation1VariableIsometry(_Ty x, _Ty h, valarray<_Ty>& y, _Ty t);
```

4. 参数

x_0：_Ty 类型变量。等距结点中的第一个结点值。

h：_Ty 类型变量。等距结点中的步长。

y：valarray<_Ty>类型引用，长度为 n。存放 n 个等距结点上的函数值。

t：_Ty 类型变量。指定插值点的值。

5. 返回值

本函数返回一个_Ty 类型函数值，即在插值点 t 处的函数近似值 $z = f(t)$。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

设函数 $y = e^{-x}$ 的列表如下：

i	1	2	3	4	5
x_i	0.1	0.2	0.3	0.4	0.5
y_i	0.904837	0.818731	0.740818	0.670320	0.606531

续表

i	6	7	8	9	10
x_i	0.6	0.7	0.8	0.9	1.0
y_i	0.548812	0.496585	0.449329	0.406570	0.367879

计算插值点 $t = 0.25, 0.63$ 和 0.95 处的函数近似值。

在使用本函数时，其中 $x_0 = 0.1, h = 0.1, n = 10$ 。

示例程序如下：

```
//Interpolation1VariableIsometry.cpp  一元全区间等距插值

#include <iostream.h>                    //输入输出流头文件
#include "Interpolation.h"                  //插值头文件
void main()
{
        double x0(0.1), h(0.1), t, s;
        double y[10] =
        {
            0.904837, 0.818731, 0.740818, 0.670320, 0.606531,
            0.548812, 0.496585, 0.449329, 0.406570, 0.367879
        };
        valarray<double> yy(y,10);

        cout << " Interpolation1VariableIsometry()" << endl << endl;

        t = 0.25;
        cout<<" t = " << t;
        s = Interpolation1VariableIsometry(x0,h,yy,t);
    cout<<"\t s = "<< s << endl;

        t = 0.63;
        cout<<" t = " << t;
        s = Interpolation1VariableIsometry(x0,h,yy,t);
    cout<<"\t s = "<< s << endl;

        t = 0.95;
        cout<<" t = " << t;
        s = Interpolation1VariableIsometry(x0,h,yy,t);
    cout<<"\t s = "<< s << endl;
}
```

8. 示例结果

```
Interpolation1VariableIsometry()

t = 0.25     s = 0.778801
```

```
t = 0.63     s = 0.532592
t = 0.95     s = 0.386741
```

9.2.3 一元三点不等距插值

1. 功能

给定 n 个不等距结点 $x_i(i=0,1,\cdots,n-1)$ 的函数值 $y_i=f(x_i)$，用抛物插值公式计算指定插值点 t 处的函数近似值 $z=f(t)$。

2. 算法

设 n 个不等距结点为 $x_0<x_1<\cdots<x_{n-1}$，其相应的函数值为 $y_i(i=0,1,\cdots,n-1)$。

为了计算指定插值点 t 处的函数近似值，选取最靠近插值点 t 的 3 个结点，即如果 $x_k<t<x_{k+1}$，则当 $|x_k-t|>|t-x_{k+1}|$ 时，取 3 个结点为 x_k,x_{k+1} 和 x_{k+2}，而当 $|x_k-t|<|t-x_{k+1}|$ 时，取 3 个结点为 x_{k-1},x_k 和 x_{k+1}，然后用抛物插值公式计算 t 点的函数近似值，即

$$z=\sum_{i=m}^{m+2}y_i\prod_{\substack{j=m\\j\neq i}}^{m+2}[(t-x_j)/(x_i-x_j)]$$

其中：

当 $|x_k-t|>|t-x_{k+1}|$ 时，$m=k$；

当 $|x_k-t|<|t-x_{k+1}|$ 时，$m=k-1$。

如果插值点 t 位于包含 n 个结点的区间外，则取区间某端的两个结点作线性插值。

3. 函数原型

```
//一元三点不等距插值
template <class _Ty>
_Ty Interpolation1Variable3PointsNotIsometry(valarray<_Ty>& x,
                                             valarray<_Ty>& y, _Ty t);
```

4. 参数

x：<_Ty>类型引用，长度为 n。存放给定的 n 个不等距结点的值。

y：valarray<_Ty>类型引用，长度为 n。存放给定的 n 个不等距结点上的函数值。

t：_Ty 类型变量。指定插值点的值。

5. 返回值

本函数返回一个_Ty 类型函数值，即在插值点 t 处的函数近似值 $z=f(t)$。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

给定列表函数如下：

x	1.615	1.634	1.702	1.828	1.921
$y=f(x)$	2.41450	2.46459	2.65217	3.03035	3.34066

用抛物插值法求 $f(x)$ 在 $x=1.682$ 和 1.813 处的函数近似值。

示例程序如下：

```
//Interpolation1Variable3PointsNotIsometry.cpp 一元三点不等距插值
```

```
#include <iomanip.h>
#include <iostream.h>                    //输入输出流头文件
#include "Interpolation.h"               //插值头文件

void main()
{
    double t,s;
    double x[5] = {1.615, 1.634, 1.702, 1.828, 1.921};
    double y[5] = {2.41450, 2.46459, 2.65271, 3.03035, 3.34066};

    valarray<double> yy(y,5);
    valarray<double> xx(x,5);

    cout << " Interpolation1Variable3PointsNotIsometry()" << endl << endl;

    t = 1.682;
    cout<<" t = " << t;
    s = Interpolation1Variable3PointsNotIsometry(xx,yy,t);
    cout << "\t s1 = " << s << endl;

    t = 1.813;
    cout<<" t = " << t;
    s = Interpolation1Variable3PointsNotIsometry(xx,yy,t);
    cout << "\t s2 = " << s << endl << endl;
}
```

8. 示例结果

```
Interpolation1Variable3PointsNotIsometry()
t = 1.682     s1 = 2.59624
t = 1.813     s2 = 2.98281
```

9.2.4 一元三点等距插值

1. 功能

给定 n 个不等距结点 $x_i = x_0 + ih(i = 0,1,\cdots,n-1)$ 的函数值 $y_i = f(x_i)$，用抛物插值公式计算指定插值点 t 处的函数近似值。

2. 算法

同本章 9.2.3 节。

3. 函数原型

```
//一元三点等距插值
template <class _Ty>
_Ty Interpolation1Variable3PointsIsometry(_Ty x0,_Ty h,
                                          valarray<_Ty>& y, _Ty t);
```

4. 参数

x_0：valarray<_Ty>类型引用，长度为 n。给定 n 个等距结点中的第一个结点值。

h:_Ty 类型变量,给定 n 等距结点的步长。

y:valarray<_Ty>类型引用,长度为 n 。存放给定 n 个等距结点上的函数值。

t:_Ty 类型变量。存放指定插值点的值。

5. 返回值

本函数返回一个_Ty 类型函数值,即在插值点 t 处的函数近似值 $z = f(t)$ 。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

设函数 $f(x) = e^{-x}$ 在 10 个等距点上的函数值如下:

x	0.1	0.2	0.3	0.4	0.5
e^{-x}	0.904837	0.818731	0.740818	0.670320	0.606531
x	0.6	0.7	0.8	0.9	1.0
e^{-x}	0.548812	0.496585	0.449329	0.406570	0.367879

计算在插值点 $t = 0.23$,0.63 和 0.95 处的函数近似值。

在使用本函数时,其中 $x_0 = 0.1, h = 0.1, n = 10$ 。

示例程序如下:

```
//Interpolation1Variable3PointsIsometry.cpp        一元三点等距插值

#include <iostream.h>                    //输入输出流头文件
#include "Interpolation.h"                  //插值头文件
void main()
{
    double x0(0.1), h(0.1), t, s;
    double y[10] =
    {
        0.904837, 0.818731, 0.740818, 0.670320, 0.606531,
        0.548812, 0.496585, 0.449329, 0.406570, 0.367879
    };
    valarray<double> yy(y,10);

    cout << " Interpolation1Variable3PointsIsometry()" << endl;

    t = 0.23;
    cout << endl << " t = " << t;
    s = Interpolation1Variable3PointsIsometry(x0,h,yy,t);
    cout << "\t s1 = " << s << endl;

    t = 0.63;
    cout<<" t = " << t;
    s = Interpolation1Variable3PointsIsometry(x0,h,yy,t);
    cout << "\t s2 = " << s << endl;
```

```
    t = 0.95;
    cout<<" t = " << t;
    s = Interpolation1Variable3PointsIsometry(x0,h,yy,t);
    cout << "\t s3 = " << s << endl;
}
```

8. 示例结果

```
Interpolation1Variable3PointsIsometry()

t = 0.23   s1 = 0.794497
t = 0.63   s2 = 0.532567
t = 0.95   s3 = 0.386716
```

9.2.5 连分式不等距插值

1. 功能

给定 n 个不等距结点 $x_i(i=0,1,\cdots,n-1)$ 的函数值 $y_i=f(x_i)$，利用连分式插值法计算指定插值点 t 处的函数近似值。

2. 算法

设给定的 n 个不等距结点为 $x_0<x_1<\cdots<x_{n-1}$，其相应的函数值为 $y_i(i=0,1,\cdots,n-1)$，则可以构造一个 n 节连分式：

$$\varphi(x)=b_0+\frac{x-x_0}{b_1}+\frac{x-x_1}{b_2}+\cdots+\frac{x-x_{n-2}}{b_{n-1}}$$

其中常数项 b_0 及部分分母 $b_0,b_1,b_2,\cdots,b_{n-1}$ 由下列递推公式计算：

$$\varphi_0(x_j)=y_j,j=0,1,\cdots,n-1$$

$$\varphi_{i+1}(x_j)=\frac{x_j-x_i}{\varphi_i(x_j)-\varphi_i(x_i)},\qquad \begin{matrix} i=0,1,\cdots,n-2 \\ j=i+1,\cdots,n-1 \end{matrix}$$

$$b_i=\varphi_i(x_i),i=0,1,\cdots,n-1$$

在实际进行插值计算时，一般在指定插值点 t 的前后各取 4 个结点就够了。此时，计算八节连分式的值 $\varphi(t)$ 作为插值点 t 处的函数近似值。

3. 函数原型

```
//连分式不等距插值
template <class _Ty>
_Ty InterpolationFractionNotIsometry(valarray<_Ty>& x,
                                     valarray<_Ty>& y, _Ty t);
```

4. 参数

x：valarray<_Ty>类型引用，长度为 n。存放给定的 n 个不等距结点值。

y：valarray<_Ty>类型引用，长度为 n。存放给定 n 个不等距结点上的函数值。

t：_Ty 类型变量。指定插值点的值。

5. 返回值

本函数返回一个_Ty 类型函数值，即在插值点 t 处的函数近似值 $z=f(t)$。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

设函数 $f(x)=\frac{1}{1+25x^2}$ 的 10 个不等距结点处的函数值如下表：

x	−1.00	−0.80	−0.65	−0.40	−0.30
$f(x)$	0.0384615	0.0588236	0.0864865	0.200000	0.307692
x	0.00	0.20	0.45	0.80	1.00
$f(x)$	1.00000	0.500000	0.164948	0.0588236	0.0384615

使用连分式插值计算 $t=-0.85$ 及 $t=0.25$ 处的函数近似值。

示例程序如下：

```
//InterpolationFractionNotIsometry.cpp  连分式不等距插值

#include <iomanip.h>
#include <iostream.h>                    //输入输出流头文件
#include "Interpolation.h"               //插值头文件

void main()
{
        double t, s;
        double x[10] =
        {
            -1.0, -0.8, -0.65, -0.4, -0.3,
            0.0,  0.2,  0.45,  0.8,  1.0
        };

  double y[10] =
        {
            0.0384615, 0.0588236, 0.0864865, 0.2,       0.307692,
            1.0,       0.5,       0.164948,  0.0588236, 0.0384615
        };

  valarray<double> yy(y,10);
        valarray<double> xx(x,10);

        cout<<"InterpolationFractionNotIsometry()"<<endl;

        t = -0.85;
        cout<<endl << " t = " << t;
        s = InterpolationFractionNotIsometry(xx,yy,t);
  cout << "\t s2 = " << s << endl;

        t = 0.25;
        cout<<" t = " << t;
```

```
        s = InterpolationFractionNotIsometry(xx,yy,t);
    cout << "\t s3 = " << s << endl;
}
```

8. 示例结果

```
InterpolationFractionNotIsometry()

t = -0.85   s1 = 0.0524591
t = 0.25    s2 = 0.390244
```

9.2.6 连分式等距插值

1. 功能

给定 n 个等距结点 $x_i = x_0 + ih(i = 0,1,\cdots,n-1)$ 的函数值 $y_i = f(x_i)$，使用连分式插值法计算指定插值点 t 处的函数近似值。

2. 算法

同本章 9.2.5 节。

3. 函数原型

```
//连分式等距插值
template <class _Ty>
_Ty InterpolationFractionIsometry(_Ty x0, _Ty h, valarray<_Ty>& y, _Ty t);
```

4. 参数

x_0：_Ty 类型变量，n 个等距结点中的第一个结点值。

h：_Ty 类型变量，n 个等距结点的步长。

y：valarray<_Ty>类型引用，长度为 n。存放 n 个等距结点上的函数值。

t：_Ty 类型变量。指定插值点的值。

5. 返回值

本函数返回一个_Ty 类型函数值，即在插值点 t 处的函数近似值 $z = f(t)$。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

设函数 $f(x) = \dfrac{1}{1+25x^2}$ 在 11 个等距点上的函数值如下：

x	−1.00	−0.80	−0.60	−0.40	−0.20	0.00
$f(x)$	0.0384615	0.0588236	0.100000	0.200000	0.500000	1.00000
x	0.20	0.40	0.60	0.80	1.00	
$f(x)$	0.500000	0.200000	0.100000	0.0588236	0.0384615	

使用连分式计算插值法计算在点 $t = -0.75$ 及 $t = -0.05$ 处的函数近似值。

在使用本函数时，其中 $x_0 = -1.00$，$h = 0.20$，$n = 11$。

示例程序如下：

```
//InterpolationFractionIsometry.cpp  连分式等距插值
```

```
#include<iomanip.h>
#include <iostream.h>                    //输入输出流头文件
#include "Interpolation.h"               //插值头文件

void main()
{
        double x0(-1.0), h(0.2), t, s;

        double y[11] =
        {
            0.0384615, 0.0588236, 0.1, 0.2, 0.5,
            1.0,       0.5,       0.2, 0.1, 0.0588236, 0.0384615
        };

        valarray<double> yy(y,11);
        cout << " InterpolationFractionIsometry() " << endl;
        t = -0.75;
        cout << endl << " t = " << t;
        s = InterpolationFractionIsometry(x0,h,yy,t);
    cout << "\t s1 = " << s << endl;

        t = -0.05;
        cout << " t = " << t;
        s = InterpolationFractionIsometry(x0,h,yy,t);
    cout << "\t s2 = " << s << endl;
}
```

8. 示例结果

```
InterpolationFractionIsometry()

t = -0.75     s1 = 0.0663901
t = -0.05     s2 = 0.941176
```

9.2.7 埃尔米特不等距插值

1. 功能

给定 n 个不等距结点 $x_i(i=0,1,\cdots,n-1)$ 的函数值 $f(x_i)$ 及一阶导数值 $f'(x_i)$，用埃尔米特(Hermite)插值公式计算指定插值点 t 处的函数近似值 $f(t)$。

2. 算法

设函数 $f(x)$ 在 n 个不等距结点 $x_0 < x_1 < \cdots < x_{n-1}$ 的函数值及一阶导数值分别为

$$y_i = f(x_i), y_i' = f'(x_i), i = 0,1,\cdots,n-1$$

则 $f(x)$ 可用埃尔米特插值多项式

$$P_{2n-1}(x) = \sum_{i=0}^{n-1} [y_i + (x - x_i)(y_i' - 2y_i l_i'(x_i))] l_i^2(x)$$

近似代替。其中：

$$l_i(x)=\prod_{\substack{j=0\\ j\neq i}}^{n-1}[(x-x_j)/(x_i-x_j)]$$

$$l_i'(x_i)=\sum_{\substack{j=0\\ j\neq i}}^{n-1}\frac{1}{x_i-x_j}$$

在实际进行插值计算时，为了减少计算工作量，用户可以适当地将远离插值点的结点抛弃。通常，只需取插值点 t 的前后各 4 个结点就够了。

3. 函数原型

```
//埃尔米特不等距插值
template <class _Ty>
_Ty InterpolationHermiteNotIsometry(valarray<_Ty>& x,
                                    valarray<_Ty>& y, valarray<_Ty>& dy, _Ty t);
```

4. 参数

x：valarray<_Ty>类型引用，长度为 n 。存放给定 n 个不等距结点的值。

y：valarray<_Ty>类型引用，长度为 n 。存放给定 n 个不等距结点上的函数值。

dy：valarray<_Ty>类型引用，长度为 n 。存放给定 n 个不等距结点上的一阶导数。

t：_Ty 类型变量。指定插值点的值。

5. 返回值

本函数返回一个_Ty 类型函数值，即在插值点 t 处的函数近似值 $z=f(t)$ 。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

设函数 $f(x)=e^{-x}$ 在 10 个不等距点上的函数值与一阶导数值如下表：

x	0.10	0.15	0.30	0.45	0.55
$f(x)$	0.904837	0.860780	0.740818	0.637628	0.576950
$f'(x)$	−0.904837	−0.860780	−0.740818	−0.637628	−0.576950
x	0.60	0.70	0.85	0.90	1.00
$f(x)$	0.548812	0.496585	0.427415	0.406570	0.367879
$f'(x)$	−0.548812	−0.496585	−0.427415	−0.406570	−0.367879

使用埃尔米特插值公式计算在插值点 $t=0.356$ 处的函数近似值 $f(t)$ 。

示例程序如下：

```
//InterpolationHermiteNotIsometry.cpp  埃米尔特不等距插值

#include <iostream.h>                  //输入输出流头文件
#include "Interpolation.h"             //插值头文件

void main()
{
        double t,s;
        double x[10] =
        {
```

```
        0.1, 0.15, 0.30, 0.45, 0.55,
        0.6, 0.7,  0.85, 0.9,  1.0
    };

    double y[10] =
    {
        0.904837, 0.860708, 0.740818, 0.637628, 0.576950,
        0.548812, 0.496585, 0.427415, 0.406570, 0.367879
    };

    valarray<double> dy(10);
    valarray<double> yy(y,10);
    valarray<double> xx(x,10);

    for(int i = 0; i < 10; i++) dy[i] = -y[i];

    cout << " InterpolationHermiteNotIsometry()" << endl;

    t = 0.356;
    cout << endl << " t = " << t;
    s = InterpolationHermiteNotIsometry(xx,yy,dy,t);
    cout << "\t s = " << s << endl;
}
```

8. 示例结果

```
InterpolationHermiteNotIsometry()

t = 0.356     s = 0.70048
```

9.2.8 埃尔米特等距插值

1. 功能

给定 n 个不等距结点 $x_i(i=0,1,\cdots,n-1)$ 的函数值 $f(x_i)$ 及一阶导数值 $f(x_i)$，使用埃尔米特(Hermite)插值公式计算指定插值点 t 处的函数近似值 $f(t)$。

2. 算法

同本章 9.2.7 节。

3. 函数原型

```
//埃尔米特等距插值
template <class _Ty>
_Ty InterpolationHermiteIsome try(_Ty x0, _Ty h,valarray<_Ty>& y, valarray<_Ty>& dy, _Ty t);
```

4. 参数

x_0：_Ty 类型变量。给定 n 个等距结点中的第一个结点。

h：_Ty 类型变量。给定 n 个等距结点的步长。

y：valarray<_Ty>类型引用，长度为 n。存放给定 n 个等距结点上的函数值。

dy:valarray<_Ty>类型引用,长度为 n 。存放给定 n 个等距结点上的一阶导数值。

t: _Ty 类型变量。指定插值点的值。

5. 返回值

本函数返回一个_Ty 类型函数值,即在插值点 t 处的函数近似值 $x = f(t)$ 。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

设函数 $f(x) = e^{-x}$ 在 10 不等距点上的函数值与一阶导数值如下表:

x	0.10	0.20	0.30	0.40	0.50
$f(x)$	0.904837	0.818731	0.740818	0.670320	0.606531
$f'(x)$	−0.904837	−0.818731	−0.740818	−0.670320	−0.606531
x	0.60	0.70	0.80	0.90	1.00
$f(x)$	0.548812	0.496585	0.449329	0.406570	0.367879
$f'(x)$	−0.548812	−0.496585	−0.449329	−0.406570	−0.367879

使用埃尔米特插值公式计算在插值点 $t = 0.752$ 处的函数近似值。

示例程序如下:

```
//InterpolationHermiteIsometry.cpp  埃米尔特等距插值

#include <iostream.h>                    //输入输出流头文件
#include "Interpolation.h"               //插值头文件

void main()
{
        double x, h, t, s;
        double y[10] =
        {
            0.904837, 0.818731, 0.740818, 0.670320, 0.606531,
            0.548812, 0.496585, 0.449329, 0.406570, 0.367879
        };

        valarray<double> yy(y,10);
        valarray<double> dy(10);

        for(int i = 0; i < 10; i++) dy[i] = -y[i];

        x = 0.1;
        h = 0.1;

        cout << " InterpolationHermiteIsometry()" << endl << endl;

        t = 0.752;
        cout << " t = " << t;
```

```
    s = InterpolationHermiteIsometry(x,h,yy,dy,t);
    cout << "\t s = " << s << endl;
}
```

8. 示例结果

```
InterpolationHermiteIsometry

t = 0.752     s = 0.471423
```

9.2.9 埃特金不等距逐步插值

1. 功能

给定 n 个不等距结点 $x_i(i=0,1,\cdots,n-1)$ 的函数值 $y_i=f(x_i)$ 及精度要求，使用埃特金(Aitken)逐步插值法计算指定插值点 t 处的函数近似值 $f(t)$。

2. 算法

设给定的 n 个不等距结点为 $x_0<x_1<\cdots<x_{n-1}$，其函数值为 $y_i(i=0,1,\cdots,n-1)$。

首先，从给定的 n 个结点中选取最靠近插值点 t 的 m 个结点 $x_0',x_1',\cdots,x_{m-1}'$，相应的函数值为 $y_0',y_1',\cdots,y_{m-1}'$，其中 $m\leqslant n$。在本函数程序中，m 的最大值为 10。

然后用此 m 个结点作埃特金逐步线性插值。

埃特金逐步线性插值的步骤如下：

(1) $y_0'\Rightarrow p_0,1\Rightarrow i$；

(2) $y_i'\Rightarrow z$

$$p_j+\frac{t-x_j'}{x_j'-x_i'}[p_j-z]\Rightarrow z,j=0,1,\cdots,i-1$$

(3) $z\Rightarrow p_i$；

(4) 若 $i=m-1$ 或 $|p_i-p_{i-1}|<\varepsilon$，则结束，$p_i$ 即为 $f(t)$ 的近似值；否则，$i+1\Rightarrow i$，转(2)。

3. 函数原型

```
//埃特金不等距逐步插值
template <class _Ty>
_Ty InterpolationAitkenNotIsometry(valarray<_Ty>& x,
                                   valarray<_Ty>& y, _Ty t, _Ty eps);
```

4. 参数

x：valarray<_Ty>类型引用，长度为 n。存放 n 个不等距结点的值。

y：valarray<_Ty>类型引用，长度为 n。存放 n 个不等距结点上的函数值。

t：_Ty 类型变量。指定插值点的值。

eps：_Ty 类型变量。插值的精度要求。

5. 返回值

本函数返回一个_Ty 类型函数值，即在插值点 t 处的函数近似值 $z=f(t)$。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

设函数 $f(x)$ 的 11 个结点上的函数值如下表：

x	−1.00	−0.80	−0.65	−0.40	−0.30	0.00
$f(x)$	0.0384615	0.0588236	0.0864865	0.200000	0.307692	1.00000
x	0.20	0.40	0.60	0.80	1.00	
$f(x)$	0.500000	0.200000	0.100000	0.0588236	0.0384615	

使用埃特金逐步线性插值计算在插值点 $t = 0.15$ 及 $t = 0.05$ 处的函数近似值。取 $eps = 10^{-6}$。

示例程序如下：

```
//InterpolationAitkenNotIsometry.cpp  埃特金不等距逐步法插值

#include <iostream.h>                 //输入输出流头文件
#include "Interpolation.h"            //插值头文件

void main()
{
        double eps = FLOATERROR, t, z;
        double x[11] =
        {
            -1.0, -0.8, -0.65, -0.4, -0.3,
             0.0,  0.2,  0.4,   0.6,  0.8, 1.0
        };

        double y[11] =
        {
            0.0384615, 0.0588236, 0.0864865, 0.2, 0.307692,
            1.0,       0.5,       0.2,       0.1, 0.0588236, 0.0384615
        };

        valarray<double> yy(y,11), xx(x,11);

        cout << " InterpolationAitkenIsometry()" << endl << endl;

        t = 0.15;
        z = InterpolationAitkenNotIsometry(xx, yy, t, eps);
        cout << " t = " << t << "\t" << "z = " << z << endl;

        t = 0.05;
        z = InterpolationAitkenNotIsometry(xx, yy, t, eps);
        cout << " t = " << t << "\t" << "z = " << z << endl;
}
```

8. 示例结果

```
InterpolationAitkenNotIsometry()

t = 0.15    z = 0.685339
```

```
t = 0.05    z = 0.959859
```

9.2.10 埃特金等距逐步插值

1. 功能

给定 n 个等距结点 $x_i = x_0 + ih(i = 0,1,\cdots,n-1)$ 的函数值 $y_i = f(x_i)$ 及精度要求，用埃特金(Aitken)逐步插值法计算指定插值点 t 处的函数近似值 $f(t)$ 。

2. 算法

算法同本章 9.2.9 节。

3. 函数原型

```
//埃特金等距逐步插值
template <class _Ty>
_Ty InterpolationAitkenIsometry(_Ty x, _Ty h,valarray<_Ty>& y, _Ty t, _Ty eps);
```

4. 参数

x： _Ty 类型变量。给定 n 个等距结点中的第一个结点值。

h： _Ty 类型变量。给定 n 个等距结点的步长。

y： valarray<_Ty>类型引用，长度为 n 。存放 n 个等距结点上的函数值。

t： _Ty 类型变量。指定插值点的值。

eps：_Ty 类型变量。插值精度的要求。

5. 返回值

本函数返回一个_Ty 类型函数值，即在插值点 t 处的函数近似值 $z = f(t)$ 。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

设函数 $f(x)$在 10 等距点上的函数值如下表：

x	0.10	0.20	0.30	0.40	0.50
$f(x)$	0.904837	0.818731	0.740818	0.670320	0.606531
x	0.60	0.70	0.80	0.90	1.00
$f(x)$	0.548812	0.496585	0.449329	0.406570	0.367879

使用埃特金逐步线性插值法计算插值点 t=0.15 及 t=0.55 处的函数近似值。取 $\varepsilon=10^{-6}$。

示例程序如下：

```
//InterpolationAitkenIsometry.cpp     埃特金等距逐步法插值

#include <iostream.h>                  //输入输出流头文件
#include "Interpolation.h"                 //插值头文件

void main(void)
{
        double eps = LONGDOUBLEERROR, t, x(0.1), h(0.1), z;

        double y[10] =
        {
```

```
        0.904837, 0.818731, 0.740818, 0.670320,0.606531,
        0.548812, 0.496585, 0.449329, 0.406570, 0.367879
    };

    valarray<double> yy(y,10);

    cout << " InterpolationAitkenIsometry()" << endl << endl;
    t = 0.15;
    z = InterpolationAitkenIsometry(x, h, yy, t, eps);
    cout << " t = " << t << "\t" << "z = " << z << endl;

    t = 0.55;
    z = InterpolationAitkenIsometry(x, h, yy, t, eps);
    cout << " t = " << t << "\t" << "z = " << z << endl;
}
```

8. 示例结果

```
InterpolationAitkenIsometry()

t = 0.15   z = 0.860709
t = 0.55   z = 0.57695
```

9.2.11 光滑不等距插值

1. 功能

给定 n 个不等距结点 $x_i(i=0,1,\cdots,n-1)$ 的函数值 $y_i=f(x_i)$，使用阿克玛(Akima)方法，计算指定区间上的三次插值多项式与指定插值点的函数值。

2. 算法

设给定的 n 个不等距结点为 $x_0<x_1<\cdots x_{n-1}$，相应的函数值为 $y_i(i=0,1,\cdots,n-1)$。

若在子区间 $[x_k,x_{k+1}](k=0,1,\cdots,n-2)$ 的两个端点处有以下 4 个条件：

$$\begin{cases} y_k=f(x_k) \\ y_{k+1}=f(x_{k+1}) \\ y_k'=g_k \\ y_{k+1}'=g_{k+1} \end{cases}$$

则在此区间上可以唯一确定一个三次多项式

$$s(x)=s_0+s_1(x-x_k)+s_2(x-x_k)^2+s_3(x-x_k)^3$$

并且就用此三次多项式计算该子区间中的插值点 t 处的函数近似值。

根据阿克玛几何条件，g_k 与 g_{k+1} 由下式计算

$$g_k=\frac{|u_{k+1}-u_k|u_{k-1}+|u_{k-1}-u_{k-2}|u_k}{|u_{k+1}-u_k|+|u_{k-1}-u_{k-2}|}$$

$$g_{k+1}=\frac{|u_{k+2}-u_{k+1}|u_k+|u_k-u_{k-1}|u_{k+1}}{|u_{k+2}-u_{k+1}|+|u_k-u_{k-1}|}$$

其中 $$u_k=\frac{y_{k+1}-y_k}{x_{k+1}-x_k}$$

并且在端点处有 $u_{-1}=2u_0-u_1, u_{-2}=2u_{-1}-u_0$

$u_{n-1}=2u_{n-2}-u_{n-3}, u_n=2u_{n-1}-u_{n-2}$

当 $u_{k+1}=u_k$ 与 $u_{k-1}=u_{k-2}$ 时

$$g_k=\frac{u_{k-1}+u_k}{2}$$

当 $u_{k+2}=u_{k+1}$ 与 $u_k=u_{k-1}$ 时

$$g_{k+1}=\frac{u_k+u_{k+1}}{2}$$

最后可以得到区间 $[x_k,x_{k+1}](k=0,1,\cdots,n-2)$ 上的三次多项式的系数为

$$s_0=y_k$$
$$s_1=g_k$$
$$s_2=(3u_k-2g_k-g_{k+1})/(x_{k+1}-x_k)$$
$$s_3=(g_{k+1}+g_k-2u_k)/(x_{k+1}-x_k)^2$$

插值点 $t\in[x_k,x_{k+1}]$ 处的函数近似值为

$$s(t)=s_0+s_1(t-x_k)+s_2(t-x_k)^2+s_3(t-x_k)^3$$

3. 函数原型

```
//光滑不等距插值
template <class _Ty>
void InterpolationSmoothNotIsometry(valarray<_Ty>& x,
                    valarray<_Ty>& y, int k, _Ty t, valarray<_Ty>& s);
```

4. 参数

x:valarray<_Ty>类型引用,长度为 n。存放给定的 n 个不等距结点值。

y:valarray<_Ty>类型引用,长度为 n。存放给定的 n 个不等距结点上的函数值。

k:整型变量。若 $k\geqslant 0$,表示只计算第 $k(k=0,1,\cdots,n-2)$ 个子区间 $[x_k,x_{k+1}]$ 上的三次多项式的系数 s_0,s_1,s_2 和 s_3(当 $k\geqslant n-1$ 时,按 $k=n-2$ 处理);若 $k<0$,表示要计算指定插值点 t 处的函数近似值 $f(t)$,并计算插值点 t 所在子区间的三次多项式的系数 s_0,s_1,s_2 和 s_3。

t:_Ty 类型变量。指定插值点的值。当 $k\geqslant 0$ 时,此形参对本函数无用,可任意。

s:valarray<_Ty>类型引用,长度为 5。其中 s_0,s_1,s_2 和 s_3 返回三次多项式的系数,s_4 返回指定插值点 t 处的函数近似值(当 $k<0$ 时)或任意(当 $k\geqslant 0$ 时)。

5. 返回值

无。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

设函数 $f(x)$ 在 11 个给定不等距结点上的函数值如下表:

x	−1.00	−0.95	−0.75	−0.55	−0.30	0.00
$f(x)$	0.0384615	0.042403	0.0663900	0.116788	0.307692	1.00000
x	0.20	0.45	0.60	0.80	1.00	
$f(x)$	0.500000	0.164984	0.100000	0.588236	0.0384615	

使用光滑插值计算指定插值点 $t=-0.85$ 及 $t=0.95$ 处的函数近似值及插值点所在子区间上的三次多项式

$$s(x)=s_0+s_1(x-x_k)+s_2(x-x_k)^2+s_3(x-x_k)^3$$

的系数 s_0,s_1,s_2 和 s_3。其中 $t\in[x_k,x_{k+1}]$。

示例程序如下：

```
//InterpolationSmoothNotIsometry.cpp  光滑不等距插值

#include <iostream.h>                 //输入输出流头文件
#include "Interpolation.h"            //插值头文件

void main()
{
        int k(-1), n(11);

        double x[11] =
        {
            -1.0, -0.95, -0.75, -0.55, -0.3,
             0.0,  0.2,   0.45,  0.60,  0.80, 1.0
        };

        double y[11] =
        {
            0.0384615, 0.0424403, 0.06639,  0.116788,  0.307692, 1.0,
            0.5,       0.164948,  0.1,      0.0588236, 0.0384615
        };

        valarray<double> xx(x,11);
        valarray<double> yy(y,11);
        valarray<double> ss(5);

        cout << " InterpolationSmoothNotIsometry()" << endl << endl;

        double t = -0.85;
        cout << " x = " << t;
        InterpolationSmoothNotIsometry(xx, yy, k, t, ss);
        cout << "\tf(x) = " << ss[4] << endl;
    cout<<" s0 = "<<ss[0]<<"\ts1 = "<<ss[1]<<"\ts2 = "<<ss[2]<<"\ts3 =
        "<<ss[3]<<endl;

        t = 0.95;
        cout << endl << " x = " << t;
        InterpolationSmoothNotIsometry(xx, yy, k, t, ss);
        cout << "\tf(x) = " << ss[4] << endl;
```

```
    cout<<" s0 = "<<ss[0]<<"\ts1 = "<<ss[1]<<"\ts2 = "<<ss[2]<<"\ts3 =
        "<<ss[3]<<endl;
}
```

8. 示例结果

```
InterpolationSmoothNotIsometry()

x = -0.85  f(x) = 0.0534042
s0 = 0.0424403  s1 = 0.0889362  s2 = 0.259985  s3 = -0.529619

x = 0.95  f(x) = 0.0417819
s0 = 0.0588236  s1 = -0.134515  s2 = 0.066864  s3 = 0.483287
```

9.2.12 光滑等距插值

1. 功能

给定 n 个等距结点 $x_i = x_0 + ih(i = 0,1,\cdots,n-1$ 的函数值 $y_i = f(x_i)$，使用阿克玛(Akima)方法，计算指定子区间的三次插值多项式与指定插值点的函数值。

2. 算法

算法同本章 9.2.11 节。

3. 函数原型

```
//光滑等距插值
template <class _Ty>
void InterpolationSmoothIsometry(_Ty x, _Ty h,
                        valarray<_Ty>& y, int k, _Ty t, valarray<_Ty>& s);
```

4. 参数

x：_Ty 类型变量。给定 n 个等距结点中的第一个结点值。

h：_Ty 类型变量。给定 n 个等距结点的步长。

y：valarray<_Ty>类型引用，长度为 n。存放 n 个等距结点上的函数值。

k：整型变量。若 $k \geqslant 0$，表示只计算第 $k(k = 0,1,\cdots,n-2)$ 个子区间 $[x_k, x_{k+1}]$ 上的三次多项式

$$s(x) = s_0 + s_1(x - x_u) + s_2(x - x_u)^2 + s_3(x - x_u)^3$$

的系数 s_0, s_1, s_2 和 s_3（当 $k \geqslant n-1$ 时，按 $k = n-2$ 处理）；若 $k < 0$，表示要计算指定插值点 t 处的函数近似值 $f(t)$，并计算插值点 t 所在子区间的三次多项式的系数 s_0, s_1, s_2 和 s_3。

t：_Ty 类型变量。指定插值点值。当 $k \geqslant 0$ 时，此形参对本函数无用，可任意。

s：valarray<_Ty>类型引用，长度为 5。其中 s_0, s_1, s_2 和 s_3 返回三次多项式的系数，s_4 返回指定插值点 t 处的函数近似值(当 $k < 0$ 时)或任意(当 $k \geqslant 0$ 时)。

5. 返回值

无。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

设函数 $f(x)$ 在 11 个等距结点上的函数值如下表：

x	−1.00	−0.80	−0.60	−0.40	−0.20	0.00
$f(x)$	0.0384615	0.0588236	0.100000	0.200000	0.500000	1.00000
x	0.20	0.40	0.60	0.80	1.00	
$f(x)$	0.500000	0.200000	0.100000	0.0588236	0.0384615	

使用光滑插值计算指定插值点 $t=-0.65$ 和 0.25 处的函数近似值及插值点所在区间的三次多项式。

$$s(x)=s_0+s_1(x-x_k)+s_2(x-x_k)^2+s_3(x-x_k)^3$$

的系数 s_0，s_1，s_2 和 s_3。其中 $t\in[x_k,x_{k+1}]$，$x_0=-1.0$，$h=0.2$。

示例程序如下：

```
//InterpolationSmoothIsometry  光滑等距插值

#include <iostream.h>       //输入输出流头文件
#include "Interpolation.h"  //插值头文件

void main()
{
        int k(-1);
        double t, x(-1.0), h(0.2);

        double y[11] =
        {
            0.0384615, 0.0588236, 0.1, 0.2, 0.5,
            1.0,       0.5,       0.2, 0.1, 0.0588236, 0.0384615
        };

        valarray<double> yy(y,11);
        valarray<double> ss(5);

        cout << " InterpolationSmoothIsometry()" << endl << endl;

        t = -0.65;
        cout << " x = " << t;
        InterpolationSmoothIsometry(x, h, yy, k, t, ss);
        cout << "\tf(x) = " << ss[4] << endl;
        cout<<" s0 = "<<ss[0]<<"\ts1 = "<<ss[1]<<"\ts2 = "<<ss[2]<<"\ts3 =
        "<<ss[3]<<endl;

        t = 0.25;
        cout << endl << " x = " << t;
        InterpolationSmoothIsometry(x, h, yy, k, t, ss);
        cout << "\tf(x) = " << ss[4] << endl;
        cout<<" s0 = "<<ss[0]<<"\ts1 = "<<ss[1]<<"\ts2 = "<<ss[2]<<"\ts3 =
```

```
        "<<ss[3]<<endl;
}
```

8. 示例结果

```
InterpolationSmoothIsometry()

x = -0.65    f(x) = 0.0882055
s0 = 0.0588236  s1 = 0.129011  s2 = 0.630092  s3 = -1.22868

x = 0.25    f(x) = 0.413068
s0 = 0.5  s1 = -1.66667  s2 = -2.19697  s3 = 15.1515
```

9.2.13 第一种边界条件的三次样条函数插值、微商与积分

1. 功能

给定 n 个结点 $x_i(i=0,1,\cdots,n-1)$ 的函数值 $y_i=f(x_i)$ 及两端点上的一阶导数值 $y_0'=f'(x_0)$，$y_{n-1}'=f'(x_{n-1})$，使用三次样条函数计算各结点的数值导数及插值区间 $[x_0,x_{n-1}]$ 的积分近似值 $s=\int_{x_0}^{x_{n-1}}f(x)\mathrm{d}x$，并对函数 $f(x)$ 进行成组插值及成组微商。

2. 算法

设给定的 n 个结点为 $x_0<x_1<\cdots<x_{n-1}$，其函数值为 $y_i=f(x_i)(i=0,1,\cdots,n-1)$，第一种边界条件为

$$\begin{cases}y_0'=f'(x_0)\\ y_{n-1}'=f'(x_{n-1})\end{cases}$$

计算其余 $n-2$ 个结点上的一阶导数值 $y_i'(i=0,1,\cdots,n-2)$ 的公式如下：

$$\begin{aligned}
&a_0=0\\
&b_0=y_0'\\
&h_j=x_{j+1}-x_j,j=0,1,\cdots,n-2\\
&\alpha_j=h_{j-1}/(h_{j-1}+h_j),j=1,2,\cdots,n-2\\
&\beta_j=3[(1-\alpha_j)(y_j-y_{j-1})/h_{j-1}+\alpha_j(y_{j+1}-y_j)/h_j],j=1,2,\cdots,n-2\\
&a_j=-\alpha_j/[2+(1-\alpha_j)a_{j-1}],j=1,2,\cdots,n-2\\
&b_j=[\beta_j-(1-\alpha_j)b_{j-1}]/[2+(1-\alpha_j)a_{j-1}],j=1,2,\cdots,n-2\\
&y_j'=a_jy_{j+1}'+b_j,j=n-2,n-3,\cdots,1
\end{aligned}$$

计算 n 个结点上的二阶导数值 $y_j''(j=0,1,\cdots,n-1)$ 的公式如下：

$$\begin{aligned}
&y_j''=6(y_{j+1}-y_j)/h_j^2-2(2y_j'+y_{j+1}')/h_j,j=0,1,\cdots,n-2\\
&y_{n-1}''=6(y_{n-2}-y_{n-1})/h_{n-2}^2+2(2y_{n-1}'+y_{n-2}')/h_{n-2}
\end{aligned}$$

利用各结点上的数值导数，并根据辛卜生(Simpson)公式，可以得到在插值区间 $[x_0,x_{n-1}]$ 上的求积公式

$$\begin{aligned}
s&=\int_{x_0}^{x_{n-1}}f(x)\mathrm{d}x\\
&=\frac{1}{2}\sum_{i=0}^{n-2}(x_{i+1}-x_i)(y_i+y_{i+1})-\frac{1}{24}\sum_{i=0}^{n-2}(x_{i+1}-x_i)^3(y_i''+y_{i+1}'')
\end{aligned}$$

利用各结点上的函数值 y_i 和一阶导数值 $y_i'(i=0,1,\cdots,n-1)$ 计算插值点 t 处的函数值、

一阶导数值及二阶导数值的公式如下（其中 $t \in [x_i, x_{i+1}]$）：

$$y(t) = \left[\frac{3}{h_i^2}(x_{i+1}-t)^2 - \frac{2}{h_i^3}(x_{i+1}-t)^3\right]y_i + \left[\frac{3}{h_i^2}(t-x_i)^2 - \frac{2}{h_i^3}(t-x_i)^3\right]y_{i+1}$$
$$+ h_i\left[\frac{1}{h_i^2}(x_{i+1}-t)^2 - \frac{1}{h_i^3}(x_{i+1}-t)^3\right]y_i' - h_i\left[\frac{1}{h_i^2}(t-x_i)^2 - \frac{1}{h_i^3}(t-x_i)^3\right]y_{i+1}'$$

$$y'(t) = \frac{6}{h_i}\left[\frac{1}{h_i^2}(x_{i+1}-t)^2 - \frac{1}{h_i}(x_{i+1}-t)\right]y_i - \frac{6}{h_i}\left[\frac{1}{h_i^2}(t-x_i)^2 - \frac{1}{h_i}(t-x_i)\right]y_{i+1}$$
$$+ \left[\frac{3}{h_i^2}(x_{i+1}-t)^2 - \frac{2}{h_i}(x_{i+1}-t)\right]y_i' + \left[\frac{3}{h_i^2}(t-x_i)^2 - \frac{2}{h_i^3}(t-x_i)\right]y_{i+1}'$$

$$y''(t) = \frac{1}{h_i^2}\left[6 - \frac{12}{h_i}(x_{i+1}-t)\right]y_i + \frac{1}{h_i^2}\left[6 - \frac{12}{h_i}(t-x_i)\right]y_{i+1}$$
$$+ \frac{1}{h_i}\left[2 - \frac{6}{h_i}(x_{i+1}-t)\right]y_i' + \frac{1}{h_i}\left[2 - \frac{6}{h_i}(t-x_i)\right]y_{i+1}'$$

3. 函数原型

```
//第一种边界条件的三次样条函数插值、微商与积分
template <class _Ty>
_Ty Interpolation3Spooling1stBoundary(valarray<_Ty>& x,
    valarray<_Ty>& y, valarray<_Ty>& dy, valarray<_Ty>& ddy,
        valarray<_Ty>& t, valarray<_Ty>& z, valarray<_Ty>& dz,
                                        valarray<_Ty>& ddz);
```

4. 参数

x：valarray<_Ty>类型引用，长度为 n。存放给定 n 个结点的值。

y：valarray<_Ty>类型引用，长度为 n。存放给定 n 个结点上的函数值。

dy：valarray<_Ty>类型引用，长度为 n，调用时，dy(0)存放给定的左端点处的一阶导数值 y_0'，dy $(n-1)$ 存放给定的右端点处一阶导数 y_{n-1}'。返回时存放 n 个给定的结点处的一阶导数值 $y_i'(i = 0,1,\cdots,n-1)$。

ddy：valarray<_Ty>类型引用，长度为 n。返回时存放 n 个给定结点处的二阶导数值 $y_i''(i = 0,1,\cdots,n-1)$。

t：valarray<_Ty>类型引用，长度为 m。返回时存放 m 个指定插值点的值。要求 $x_0 < t(j) < x_{n-1}(j = 0,1,\cdots,m-1)$。

z：valarray<_Ty>类型引用，长度为 m。返回时存放 m 个指定插值点处的函数值。

dz：valarray<_Ty>类型引用，长度为 m。返回时存放 m 个指定插值点处的一阶导数值。

ddz：valarray<_Ty>类型引用，长度为 m。返回时存放 m 个指定插值点处的二阶导数值。

5. 返回值

本函数返回一个_Ty 类型的积分值

$$\int_{x_0}^{x_{n-1}} f(x)\mathrm{d}x$$

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

已知某直升飞机旋转机翼外形曲线上的部分坐标值如下：

x	0.52	8.0	17.95	28.65	50.65	104.6
$f(x)$	5.28794	13.8400	20.2000	24.9000	31.1000	36.5000
x	156.6	260.7	364.4	486.0	507.0	520.0
$f(x)$	36.6000	31.0000	20.9000	7.80000	1.50000	0.200000

且两端点的一阶导数值为 $y'_0 = 1.86548, y'_{n-1} = -0.046115$ 。

计算各结点处的一阶及二阶导数值，在区间 [0.52,520.0] 内的积分值，并计算在 8 个插值点 4.0,14.0,30.0,60.0,130.0,230.0,450.0 和 515.0 处的函数值，一阶导数值及二阶导数值。

在本例中，$n = 12, m = 8$ 。

示例程序如下：

```
//Interpolation3SpoolinglstBoundary.cpp
//第一种边界条件的三次样条函数插值，微商，积分

#include <iostream.h>                    //输入输出流头文件
#include "Interpolation.h"               //插值头文件

void main()
{
    int m(8), n(12);
        double s, dy[12];
        double x[12] =
        {
          0.52,    8.0, 17.95, 28.65, 50.65, 104.6,
          156.6, 260.7, 364.4, 468.0, 507.0, 520.0
        };
        double y[12] =
        {
            5.28794, 13.84, 20.2, 24.9, 31.1, 36.5,
            36.6,  31.0, 20.9,  7.8,  1.5,  0.2
        };
        double t[8] =
        {
            4.0, 14.0, 30.0, 60.0, 130.0, 230.0, 450.0, 515.0
        };
        dy[0] = 1.86548;
        dy[11] = -0.046115;

        valarray<double> xx(x, 12);
        valarray<double> yy(y, 12);
        valarray<double> dyy(dy, 12);
        valarray<double> ddyy(12);
        valarray<double> tt(t, 8);
```

```
        valarray<double> zz(8);
        valarray<double> dzz(8);
        valarray<double> ddzz(8);

        cout << "Interpolation3Spooling1stBoundary()" << endl << endl;
        s = Interpolation3Spooling1stBoundary(xx, yy, dyy, ddyy, tt, zz, dzz, ddzz);

    cout << "x(i)        y(i)        dy(i)        ddy(i)" <<
        endl;
    for(int i = 0; i<12; i + + )
      cout<<endl<<xx[i]<<"\t\t"<<yy[i]<<"\t\t"<<dyy[i]<<"\t\t"<<ddyy[i];

    cout << endl << endl << "s = " << s << endl << endl;

    cout<<"t(i)        z(i)        dz(i)
  ddz(i)"<<endl;
    for(i = 0; i<8; i + + )
          cout<<endl<<tt[i]<<"\t\t"<<zz[i]<<"\t\t"<<dzz[i]<<"\t\t"<<ddzz
        [i];

         cout << endl;
}
```

8. 示例结果

```
Interpolation3Spooling1stBoundary()

x(i)        y(i)        dy(i)        ddy(i)
0.52        5.28794     1.86548      -0.279319
8           13.84       0.743662     -0.0206327
17.95       20.2        0.532912     -0.0217292
28.65       24.9        0.368185     -0.00906092
50.65       31.1        0.208755     -0.00543268
104.6       36.5        0.0293142    -0.00121944
156.6       36.6        -0.0211538   -0.000721641
260.7       31          -0.0815142   -0.00043802
364.4       20.9        -0.106449    -4.28883e-005
468         7.8         -0.164223    -0.00107244
507         1.5         -0.135256    0.00255796
520         0.2         -0.046115    0.011156

s = 12904.4

t(i)        z(i)        dz(i)        ddz(i)
4           10.3314     1.10286      -0.158967
14          17.9266     0.617882     -0.0212939
```

30	25.3889	0.356103	-0.00883827
60	32.825	0.161373	-0.00470249
130	36.8774	0.00142856	-0.000976284
230	33.2829	-0.0667831	-0.000521662
450	10.5919	-0.146529	-0.000893563
515	0.556246	-0.0936277	0.00784907

9.2.14 第二种边界条件的三次样条函数插值、微商与积分

1. 功能

给定 n 个结点 $x_i(i=0,1,\cdots,n-1)$ 的函数值 $y_i=f(x_i)$ 及两端点的二阶导数值 $y_0''=f''(x_0)$，$y_{n-1}''=f''(x_{n-1})$ 使用三次样条函数计算各结点上的数值导数 y_i'，$y_i''(i=0,1,\cdots,n-1)$ 及插值区间 $[x_0,x_{n-1}]$ 的积分近似值 $s=\int_{x_0}^{x_{n-1}}f(x)\mathrm{d}x$，并对函数 $f(x)$ 进行成组插值及成组微商。

2. 算法

设给定的 n 个结点为 $x_0<x_1<\cdots<x_{n-1}$，其函数值为 $y_i(i=0,1,\cdots,n-1)$，第二种边界条件为

$$\begin{cases}y_0''=f''(x_0)\\ y_{n-1}''=f''(x_{n-1})\end{cases}$$

计算 n 个结点上的一阶导数值 $y_i'(i=0,1,\cdots,n-1)$ 的公式如下：

$$a_0=-0.5$$

$$b_0=3\cdot\frac{y_1-y_0}{2(x_1-x_0)}-y_0''(x_1-x_0)/4$$

$$h_j=x_{j+1}-x_j,j=0,1,\cdots,n-2$$

$$\alpha_j=h_{j-1}/(h_{j-1}+h_j),j=1,2,\cdots,n-2$$

$$\beta_j=3[(1-\alpha_j)(y_j-y_{j-1})/h_{j-1}+\alpha_j(y_{j+1}-y_j)/h_j],j=1,2,\cdots,n-2$$

$$a_j=-\alpha_j/[2+(1-\alpha_j)a_{j-1}],j=1,2,\cdots,n-2$$

$$b_j=[\beta_j-(1-\alpha_j)b_{j-1}]/[2+(1-\alpha_j)a_{j-1}],j=1,2,\cdots,n-2$$

$$y_{n-1}'=[3(y_{n-1}-y_{n-2})/h_{n-2}+y_{n-1}''h_{n-2}/2-b_{n-2}]/(2+a_{n-2})$$

$$y_j'=a_jy_{j+1}'+b_j,j=n-2,n-3,\cdots,0$$

计算 n 个结点上的二阶导数值 $y_j''(j=0,1,\cdots,n-1)$，插值区间 $[x_0,x_{n-1}]$ 上的积分值，计算插值点 t 处的函数值、一阶导数值及二阶导数值的公式同 5.13 节。

3. 函数原型

```
//第二种边界条件的三次样条函数插值、微商与积分
template <class _Ty>
_Ty Interpolation3Spooling2ndBoundary(valarray<_Ty>& x,
    valarray<_Ty>& y, valarray<_Ty>& dy, valarray<_Ty>& ddy,
        valarray<_Ty>& t, valarray<_Ty>& z, valarray<_Ty>& dz,
                                            valarray<_Ty>& ddz);
```

4. 参数

x：　valarray<_Ty>类型引用，长度为 n。存放 n 个给定结点的值。

y： valarray<_Ty>类型引用，长度为 n 。存放 n 个给定结点上的函数值。

dy：valarray<_Ty>类型引用，长度为 n ，返回时存放 n 个给定的结点上的一阶导数值。

ddy：valarray<_Ty>类型引用，长度为 n ，调用时，ddy(0)存放给定的左端点处的二阶导数值 y_0'' ，ddy $(n-1)$ 存放给定的右端点处二阶导数 y_{n-1}'' 。返回时存放 n 个给定的结点处的二阶导数值 $y_i''(i=0,1,\cdots,n-1)$ 。

t：valarray<_Ty>类型引用，长度为 m 。返回时存放 m 个指定插值点的值。要求 $x_0 < t(j) < x_{n-1}(j=0,1,\cdots,m-1)$ 。

z：valarray<_Ty>类型引用，长度为 m 。返回时存放 m 个指定插值点处的函数值。

dz：valarray<_Ty>类型引用，长度为 m 。返回时存放 m 个指定插值点处的一阶导数值。

ddz：valarray<_Ty>类型引用，长度为 m 。返回时存放 m 个指定插值点处的二阶导数值。

5. 返回值

本函数返回一个_Ty类型积分值

$$\int_{x_0}^{x_{n-1}} f(x)\mathrm{d}x$$

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

同本章 9.2.13 节的例，其中边界条件改为

$$y_0'' = -0.279319, y_{n-1}'' = -0.0111560$$

示例程序如下：

```
//Interpolation3Spooling2ndBoundary.cpp  第二种边界条件的三次样条函数插值，微商，积分

#include <iostream.h>                    //输入输出流头文件
#include "Interpolation.h"               //插值头文件

void main()
{
        int m(8), n(12);
        double s, ddy[12];

        double x[12] =
        {
            0.52,   8.0, 17.95, 28.65, 50.65, 104.6,
           156.6, 260.7, 364.4, 468.0, 507.0, 520.0
        };

        double y[12] =
        {
           5.28794, 13.84, 20.2, 24.9, 31.1, 36.5,
             36.6,  31.0, 20.9,  7.8,  1.5,  0.2
        };
```

```
        double t[8] = {4.0, 14.0, 30.0, 60.0, 130.0, 230.0, 450.0, 515.0};

        ddy[0] = -0.279319;
        ddy[11] = 0.011156;

        valarray<double> yy(y, 12);
        valarray<double> dyy(12);
        valarray<double> ddyy(ddy, 12);
        valarray<double> zz(8);
        valarray<double> dzz(8);
        valarray<double> ddzz(8);
  valarray<double> tt(t, 8);
        valarray<double> xx(x, 12);

        s = Interpolation3Spooling2ndBoundary(xx, yy, dyy, ddyy, tt, zz, dzz, ddzz);
        cout<<"Interpolation3Spooling2ndBoundary()"<<endl<<endl;

        cout<<"x(i)          y(i)          dy(i)
        ddy(i)"<<endl;
        for(int i = 0; i <12; i + + )

            cout<<endl<<xx[i]<<"\t\t"<<yy[i]<<"\t\t"<<dyy[i]<<"\t\t"<<ddyy
        [i];

            cout << endl << endl << "s = " << s <<endl << endl;

        cout<<"t(i)              z(i)              dz(i)
        ddz(i)"<<endl;
  for(i = 0; i<8; i + + )

            cout<<endl<<tt[i]<<"\t\t"<<zz[i]<<"\t\t"<<dzz[i]<<"\t\t"<<ddzz
        [i];

        cout << endl;
}
```

8. 示例结果

```
Interpolation3Spooling2ndBoundary()

x(i)          y(i)          dy(i)          ddy(i)

0.52          5.28794       1.86548        -0.279319
8             13.84         0.743662       -0.0206326
17.95         20.2          0.532912       -0.0217292
```

28.65	24.9	0.368185	-0.00906091
50.65	31.1	0.208755	-0.00543268
104.6	36.5	0.0293142	-0.00121944
156.6	36.6	-0.0211538	-0.000721641
260.7	31	-0.0815142	-0.00043802
364.4	20.9	-0.106449	-4.28882e-005
468	7.8	-0.164223	-0.00107244
507	1.5	-0.135256	0.00255796
520	0.2	-0.0461151	0.011156

s = 12904.4

t(i)	z(i)	dz(i)	ddz(i)
4	10.3314	1.10286	-0.158968
14	17.9266	0.617882	-0.0212939
30	25.3889	0.356103	-0.00883827
60	32.825	0.161373	-0.00470249
130	36.8774	0.00142856	-0.000976284
230	33.2829	-0.0667831	-0.000521662
450	10.5919	-0.146529	-0.000893563
515	0.556247	-0.0936277	0.00784906

9.2.15 第三种边界条件的三次样条函数插值、微商与积分

1. 功能

给定 n 个结点 $x_i(i=0,1,\cdots,n-1)$ 的函数值 $y_i=f(x_i)$ 及第三种边界条件，使用三次样条函数，计算各给定结点的一阶导数值 y_i'、二阶导数值 $y_i''(i=0,1,\cdots,n-1)$ 及插值区间 $[x_0,x_{n-1}]$ 上的积分近似值 $s=\int_{x_0}^{x_{n-1}} f(x)\mathrm{d}x$，并对函数 $f(x)$ 进行成组插值及成组微商。

2. 算法

设给定的 n 个结点为 $x_0<x_1<\cdots<x_{n-1}$，其的函数值为 $y_i(i=0,1,\cdots,n-1)$，第三种边界条件为

$$y_0=y_{n-1},y_0'=y_{n-1}',y_0''=y_{n-1}''$$

计算 n 个结点上的一阶导数值 $y_j'(j=0,1,\cdots,n-2)$ 的公式如下：

$$a_0=0,b_0=1,c_1=0$$

$$h_0=h_{n-1},y_0-y_{-1}=y_{n-1}-y_{n-2}$$

$$h_j=x_{j+1}-x_j,j=0,1,\cdots,n-2$$

$$\alpha_j=h_{j-1}/(h_{j-1}+h_j),j=1,2,\cdots,n-2$$

$$\beta_j=3[(1-\alpha_j)(y_j-y_{j-1})/h_{j-1}+\alpha_j(y_{j+1}-y_j)/h_j],j=1,2,\cdots,n-2$$

$$a_j=-\alpha_{j-1}/[2+(1-x_{j-1})a_{j-1}],j=1,2,\cdots,n-2$$

$$b_j=-(1-a_{j-1})b_{j-1}/[2+(1-\alpha_{j-1})a_{j-1}],j=1,2,\cdots,n-2$$

$$c_j=[\beta_{j-1}-(1-\alpha_{j-1})c_{j-1}]/[2+(1-\alpha_{j-1})a_{j-1}],j=1,2,\cdots,n-2$$

$$u_{n-1}=1,v_{n-1}=0$$

$$u_j=a_ju_{j+1}+b_j,j=n-2,n-3,\cdots,1$$

$$v_j=a_jv_{j+1}+c_j,j=n-2,n-3,\cdots,1$$

$$y'_{n-2}=[\beta_{n-2}-\alpha_{n-2}v_1-(1-\alpha_{n-2})v_{n-2}]/[2+\alpha_{n-2}u_1+(1-\alpha_{n-2})u_{n-2}]$$

$$y'_j=u_{j+1}y'_{n-2}+v_{j+1},j=1,2,\cdots,n-3$$

$$y'_{n-1}=y'_0$$

计算 n 个给定结点上的二阶导数值 $y''_j(j=0,1,\cdots,n-1)$，插值区间$[x_0,x_{n-1}]$上的积分值，计算插值点 t 处的函数值、一阶导数值及二阶导数值的公式同本章 9.2.13 节。

3. 函数原型

```
//第三种边界条件的三次样条函数插值、微商与积分
template <class _Ty>
_Ty Interpolation3Spooling3thBoundary(valarray<_Ty>& x,
    valarray<_Ty>& y, valarray<_Ty>& dy, valarray<_Ty>& ddy,
      valarray<_Ty>& t, valarray<_Ty>& z, valarray<_Ty>& dz,
                                        valarray<_Ty>& ddz);
```

4. 参数

x：valarray<_Ty>类型引用，长度为 n。存放 n 个给定结点的值。

y：valarray<_Ty>类型引用，长度为 n。存放 n 个给定结点上的函数值。要求 $y_0=y_{n-1}$。

n：整型变量。给定结点的个数。

dy：valarray<_Ty>类型引用，长度为 n，返回时存放 n 个给定结点上的一阶导数值。

ddy：valarray<_Ty>类型引用，长度为 n，返回时存放 n 个给定结点上的二阶导数值。

t：valarray<_Ty>类型引用，长度为 m。存放 m 个指定插值点的值。

m：整型变量。指定插值点的个数。

z：valarray<_Ty>类型引用，长度为 m。返回时存放 m 个指定插值点处的函数值。

dz：valarray<_Ty>类型引用，长度为 m。返回时存放 m 个指定插值点处的一阶导数值。

ddz：valarray<_Ty>类型引用，长度为 m。返回时存放 m 个指定插值点处的二阶导数值。

5. 返回值

本函数返回一个_Ty 类型积分值

$$\int_{x_0}^{x_{n-1}}f(x)\mathrm{d}x$$

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

给定间隔为 10°的 $\sin x$ 函数表，使用三次样条插值计算间隔为 5°的 $\sin x$ 函数表，并计算其一阶、二阶导数值以及在一个周期内的积分值。

在本函数中 $n=37,m=36$。

示例程序如下：

```
//Interpolation3Spooling3thBoundary.cpp  第三种边界条件的三次样条函数插值,微商,积分
#include <iostream.h>                    //输入输出流头文件
#include "Interpolation.h"               //插值头文件
```

```
void main()
{
        int m(36), n(37), j(0);
        double u, s;

        double x[37], y[37], t[36];

        for(int i = 0; i<37; i + + )
        {
          x[i] = i * 6.2831852 / 36.0;
          y[i] = sin(x[i]);
        }

        for(i = 0; i<36; i + + )
        {
          t[i] = (0.5 + i) *  6.2831852 / 36.0;
        }

        valarray<double> yy(y, 37);
        valarray<double> dyy(37);
        valarray<double> ddyy(37);
        valarray<double> zz(36);
        valarray<double> dzz(36);
        valarray<double> ddzz(36);
valarray<double> tt(t, 36);
        valarray<double> xx(x, 37);

        s = Interpolation3Spooling3thBoundary(xx, yy, dyy, ddyy, tt, zz, dzz, ddzz);

        cout << "Interpolation3Spooling3thBoundary()" << endl << endl;

        cout<<"x(i)          y(i) = sin(x)              dy(i) = cos(x)
        ddy(i) = - sin(x)"<<endl;
        cout<<endl<<x[0]<<"\t\t"<<yy[0]<<"\t\t"<<dyy[0]<<"\t\t"<<ddyy[0];

        for(i = 0; i<=35; i + + )
        {
          u = t[i] * 36.0 / 0.62831852;
          cout<<endl<<u<<"\t\t"<<zz[i]<<"\t\t"<<dzz[i]<<"\t\t"<<ddzz
    [i];

          u = x[i+1] * 36.0 / 0.62831852;
          j + + ;
          cout<<endl<<u<<"\t\t"<<yy[j]<<"\t\t"<<dyy[j]<<"\t\t"<<ddyy
```

```
    [j];
        }
        cout << endl << endl << "s = " << s << endl << endl;
}
```

8. 示例结果

Interpolation3Spooling3thBoundary()

x(i)	y(i) = sin(x)	dy(i) = cos(x)	ddy(i) = - sin(x)
0	0	0.999995	5.3726e-008
5	0.0871555	0.996197	-0.0870447
10	0.173648	0.984803	-0.174089
15	0.258818	0.965928	-0.258489
20	0.34202	0.939688	-0.342889
25	0.422617	0.90631	-0.42208
30	0.5	0.866021	-0.501271
35	0.573575	0.819154	-0.572846
40	0.642788	0.76604	-0.644421
45	0.707105	0.707108	-0.706206
50	0.766044	0.642784	-0.767991
55	0.81915	0.573578	-0.818108
60	0.866025	0.499997	-0.868226
65	0.906306	0.422619	-0.905153
70	0.939693	0.342018	-0.94208
75	0.965923	0.25882	-0.964695
80	0.984808	0.173647	-0.98731
85	0.996192	0.087156	-0.994926
90	1	2.67948e-008	-1.00254
95	0.996192	-0.0871559	-0.994926
100	0.984808	-0.173647	-0.98731
105	0.965923	-0.25882	-0.964695
110	0.939693	-0.342018	-0.94208
115	0.906306	-0.422619	-0.905153
120	0.866025	-0.499997	-0.868226
125	0.81915	-0.573578	-0.818109
130	0.766044	-0.642784	-0.767991
135	0.707105	-0.707108	-0.706206
140	0.642788	-0.76604	-0.644421
145	0.573575	-0.819154	-0.572846
150	0.5	-0.866021	-0.501271
155	0.422617	-0.90631	-0.42208
160	0.34202	-0.939688	-0.342889
165	0.258818	-0.965928	-0.258489
170	0.173648	-0.984803	-0.174089

175	0.0871556	-0.996197	-0.0870448
180	5.35898e-008	-0.999995	-5.3726e-008
185	-0.0871555	-0.996197	0.0870447
190	-0.173648	-0.984803	0.174089
195	-0.258818	-0.965928	0.258489
200	-0.34202	-0.939688	0.342889
205	-0.422617	-0.90631	0.42208
210	-0.5	-0.866021	0.50127
215	-0.573575	-0.819154	0.572846
220	-0.642788	-0.766041	0.644421
225	-0.707105	-0.707108	0.706206
230	-0.766044	-0.642784	0.767991
235	-0.81915	-0.573578	0.818108
240	-0.866025	-0.499997	0.868226
245	-0.906306	-0.422619	0.905153
250	-0.939693	-0.342018	0.94208
255	-0.965923	-0.25882	0.964695
260	-0.984808	-0.173647	0.98731
265	-0.996192	-0.087156	0.994926
270	-1	-8.03843e-008	1.00254
275	-0.996192	0.0871558	0.994926
280	-0.984808	0.173647	0.98731
285	-0.965923	0.25882	0.964695
290	-0.939693	0.342018	0.94208
295	-0.906306	0.422619	0.905153
300	-0.866025	0.499997	0.868226
305	-0.81915	0.573578	0.818109
310	-0.766045	0.642784	0.767991
315	-0.707105	0.707108	0.706206
320	-0.642788	0.76604	0.644421
325	-0.573575	0.819154	0.572846
330	-0.5	0.866021	0.501271
335	-0.422617	0.90631	0.42208
340	-0.34202	0.939688	0.342889
345	-0.258819	0.965928	0.258489
350	-0.173648	0.984803	0.17409
355	-0.0871556	0.996197	0.0870448
360	-1.0718e-007	0.999995	5.3726e-008

s = 5.63438e-015

9.2.16 二元三点插值

1. 功能

给定矩形域上 $n\times m$ 个结点 $(x_i,y_j)(i=0,1,\cdots,n-1;j=0,1,\cdots,m-1)$ 的函数值 $z_{ij}=$

$z(x_i,y_i)$，使用二元三点插值公式计算指定插值点(u,v)处的函数近似值$\omega=z(u,v)$。

2. 算法

设给定矩形域上的$n\times m$个结点在两个方向上的坐标分别为

$$x_0<x_1<\cdots<x_{n-1}$$

及

$$y_0<y_1<\cdots<y_{m-1}$$

相应的函数值为

$$z_{ij}=z(x_i,y_i),i=0,1,\cdots,n-1;j=0,1,\cdots,m-1$$

选最近插值点(u,v)的9个结点，其两个方向上的坐标分别为

$$x_p<x_{p+1}<x_{p+2}$$

及

$$y_q<y_{q+1}<y_{q+2}$$

然后用二元三点插值公式

$$z(x,y)=\sum_{i=p}^{p+2}\sum_{j=q}^{q+2}\left(\prod_{\substack{k=p\\k\neq i}}^{p+2}\frac{x-x_k}{x_i-x_k}\right)\left(\prod_{\substack{l=q\\l\neq j}}^{q+2}\frac{y-y_l}{y_j-y_l}\right)z_{ij}$$

计算插值点(u,v)处的函数近似值。

3. 函数原型

```
//二元三点插值
template <class _Ty>
_Ty Interpolation2Variable3Points(valarray<_Ty>& x, valarray<_Ty>& y,
                                  matrix<_Ty> z, _Ty u, _Ty v);
```

4. 参数

x:valarray<_Ty>类型引用，长度为n。存放给定$n\times m$个结点x方向的n个坐标。

y:valarray<_Ty>类型引用，长度为m。存放给定$n\times m$个结点y方向的m个坐标。

z:matrix<_Ty>类型引用，体积为$n\times m$。存放给定$n\times m$个结点上的函数值。

$$z_{ij}=z(x_i,y_i),i=0,1,\cdots,n-1;j=0,1,\cdots,m-1$$

u:_Ty类型变量。指定插值点的x方向坐标。

v:_Ty类型变量。指定插值点的y方向坐标。

5. 返回值

本函数返回一个_Ty类型函数值$f(u,v)$。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

设二元函数

$$z(x,y)=e^{-(x-y)}$$

取以下6×5个结点：

$$x_i=0.2i,i=0,1,\cdots,5$$

$$y_j=0.25j,j=0,1,\cdots,4$$

其函数值为

$$z(i,j)=e^{-(x_i-y_j)},i=0,1,\cdots,5,j=0,1,\cdots,4$$

使用二元三点插值法计算插值点(0.9,0.8)及(0.3,0.9)处的函数近似值。

示例程序如下：

```
//Interpolation2Variable3Points.cpp  二元三点插值

# include <iostream.h>              //输入输出流头文件
# include "Interpolation.h"         //插值头文件

void main()
{
        int  i, j;
        double u, v, w;

        valarray<double> x(6), y(5);

        for(i=0; i<6; i++) x[i] = 0.2 * i;
        for(i=0; i<5; i++) y[i] = 0.25 * i;

        matrix<double> z(6,5);

        for(i = 0; i < 6; i++)
          for(j = 0; j < 5; j++)
          {
               z(i,j) = exp( -(x[i] - y[j]) );
          }

         cout << "Interpolation2Variable3Points()" << endl << endl;

         u = 0.90;
         v = 0.80;
         w = Interpolation2Variable3Points(x, y, z, u, v);
  cout << "x = 0.9    y = 0.8          z(x,y) = " << w << endl;

         u = 0.3;
         v = 0.9;
         w = Interpolation2Variable3Points(x, y, z, u, v);
  cout << "x = 0.3    y = 0.9          z(x,y) = " << w << endl;
}
```

8. 示例结果

```
Interpolation2Variable3Points()

x = 0.9    y = 0.8          z(x,y) = 0.904786
x = 0.3    y = 0.9          z(x,y) = 1.82277
```

9.2.17 二元全区间插值

1. 功能

给定矩形域上 $n\times m$ 个结点 $(x_i,y_j)(i=0,1,\cdots,n-1;j=0,1,\cdots,m-1)$ 的函数值 $z_{ij}=$

$z(x_i,y_i)$，使用二元三点插值公式计算指定插值点(u,v)处的函数近似值$\omega=z(u,v)$。

2. 算法

设给定矩形域上的$n\times m$个结点在两个方向上的坐标分别为

$$x_0<x_1<\cdots<x_{n-1}$$

及

$$y_0<y_1<\cdots<y_{m-1}$$

相应的函数值为

$$z_{ij}=z(x_i,y_i),i=0,1,\cdots,n-1;j=0,1,\cdots,m-1$$

计算插值点(u,v)处的函数值$\omega=z(u,v)$。

以插值点(u,v)为中心，在x方向上，前后各取4个坐标

$$x_p<x_{p+1}<x_{p+2}<x_{p+3}<u<x_{p+4}<x_{p+5}<x_{p+6}<x_{p+7}$$

在y方向上，前后也各取4个坐标

$$y_q<y_{q+1}<y_{q+2}<y_{q+3}<v<y_{q+4}<y_{q+5}<y_{q+6}<y_{q+7}$$

然后用二元插值公式

$$z(x,y)=\sum_{i=p}^{p+7}\sum_{j=q}^{q+7}\left(\prod_{\substack{k=p\\k\neq i}}^{p+7}\frac{x-x_k}{x_i-x_k}\right)\left(\prod_{\substack{l=q\\l\neq j}}^{q+7}\frac{y-y_l}{y_j-y_l}\right)z_{ij}$$

计算插值点(u,v)处的函数近似值。

3. 函数原型

```
//二元全区间插值
template <class _Ty>
_Ty Interpolation2VariableWholeInterval(valarray<_Ty>& x,
                                        valarray<_Ty>& y, matrix<_Ty> z, _Ty u, _Ty v);
```

4. 参数

x：valarray<_Ty>类型引用，长度为n。存放给定$n\times m$个结点x方向的n个坐标。

y：valarray<_Ty>类型引用，长度为m。存放给定$n\times m$个结点y方向的m个坐标。

z：matrix<_Ty>类型引用，体积为$n\times m$。存放给定$n\times m$个结点上的函数值。

$$z_{ij}=z(x_i,y_i),i=0,1,\cdots,n-1;j=0,1,\cdots,m-1$$

u：_Ty类型变量。指定插值点的x方向坐标。

v：_Ty类型变量。指定插值点的y方向坐标。

5. 返回值

本函数返回一个_Ty类型函数值$f(u,v)$。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

设二元函数

$$z(x,y)=e^{-(x-y)}$$

取以下11×11个结点：

$$x_i=0.1i,i=0,1,\cdots,10$$

$$y_j=0.1j,j=0,1,\cdots,10$$

其函数值为

$$z(i,j)=e^{-(x_i-y_j)},i=0,1,\cdots,10,j=0,1,\cdots,10$$

使用二元插值计算法计算插值点(0.35,0.65)及(0.45,0.55)处的函数近似值。

示例程序如下：

```
//Interpolation2VariableWholeInterval.cpp  二元全区间插值

#include <iostream.h>              //输入输出流头文件
#include "Interpolation.h"         //插值头文件

void main()
{
        int  i, j;
        double u, v, w;
        valarray<double> x(11), y(11);

        for(i=0; i<11; i++)
        {
          x[i] = 0.1 * i;
          y[i] = x[i];
         }
         matrix<double> z(11,11);

         for(i=0; i<11; i++)
         {
           for(j=0; j<11; j++)
           {
            z(i,j) = exp( -(x[i] - y[j]) );
           }
         }

         cout << "Interpolation2VariableWholeInterval()" << endl << endl;

         u = 0.35;v = 0.65;
         w = Interpolation2VariableWholeInterval(x, y, z, u, v);
  cout << "x = 0.35     y = 0.65          z(x,y) = " << w << endl;

         u = 0.45;v = 0.55;
         w = Interpolation2VariableWholeInterval(x, y, z, u, v);
  cout << "x = 0.45     y = 0.55          z(x,y) = " << w << endl;
}
```

8. 示例结果

```
Interpolation2VariableWholeInterval()

x = 0.35      y = 0.65          z(x,y) = 1.34986
x = 0.45      y = 0.55          z(x,y) = 1.10517
```

第10章 数值积分

本章介绍数值积分的算法，每种算法都用C++语言以函数的形式实现。10.1节介绍数值积分的基本概念，以及数值积分算法程序原型的头文件。10.2节给出了计算各种数值积分值的使用示例及相关说明。

10.1 数值积分与头文件

本节介绍数值积分基本概念，并包括数值积分算法的函数程序原型，这些原型都包含在头文件 Integral.h 中，函数的定义内容在 Integral.inl 文件中。

本章介绍的数值积分算法及程序包括：

- 变步长梯形法求积；
- 变步长辛卜生法求积；
- 自适应梯形法求积；
- 龙贝格法求积；
- 一维连分式法求积；
- 高振荡函数法求积；
- 勒让德—高斯法求积；
- 拉盖尔—高斯法求积；
- 埃尔米特—高斯法求积；
- 切比雪夫法求积；
- 蒙特卡洛法求积；
- 二重变步长辛卜生法求积；
- 多重高斯法求积；
- 二重连分式法求积；
- 多重蒙特卡洛法求积。

10.1.1 基本概念

1. 数值积分

数值积分就是在积分数值计算中，利用诸如插值多项式这类“简单”函数来近似代替积分中的一般被积函数，然后对这些“简单”被积函数进行积分，推导出相关的积分生成公式，再用这些生成的公式去近似计算其他相对复杂、“困难”的被积函数的积分，计算出其积分值。

2. 数值积分公式

利用插值多项式这类“简单”函数来近似代替积分中的一般被积函数，对其进行积分，推导出相关的积分生成公式，这些公式就称为“数值积分公式”，也简称为“积分公式”。

10.1.2 Integral.h

```
// Integral.h  数值积分头文件
// Ver 1.0.0.0
// 版权所有(C) 2002
// 版权所有(C) 何渝，2002
// 最后修改：2002.8.26.

#ifndef _INTEGRAL_H          //避免多次编译
#define _INTEGRAL_H

#include <comm.h>            //公共头文件
#include <random.h>          //随机算法头文件
#include <valarray>          //模板类 valarray 的标准头文件
using namespace std;         //名字空间

//template <class _Ty = float>

//变步长梯形法求积
template <class _Ty>
_Ty IntegralTrapezia(_Ty a, _Ty b, _Ty eps);

//变步长辛卜生法求积
template <class _Ty>
_Ty IntegralSimpson1D(_Ty a, _Ty b, _Ty eps);

//自适应梯形法求积
template <class _Ty>
_Ty IntegralTrapeziaSelfAdapt(_Ty a, _Ty b, _Ty eps, _Ty d);

//龙贝格法求积
template <class _Ty>
_Ty IntegralRomberg(_Ty a, _Ty b, _Ty eps);

//一维连分式法求积
template <class _Ty>
_Ty IntegralFraction1D(_Ty a, _Ty b, _Ty eps);

//高振荡函数法求积
template <class _Ty>
void IntegralSurge(_Ty a, _Ty b, int m, valarray<_Ty>& fa,
                   valarray<_Ty>& fb, valarray<_Ty>& s);
//勒让德-高斯法求积
template <class _Ty>
```

```
_Ty IntegralLegendreGauss(_Ty a, _Ty b, _Ty eps);

//拉盖尔—高斯法求积
template <class _Ty>
void IntegralLaguerreGauss(_Ty& dValue);

//埃尔米特—高斯法求积
template <class _Ty>
void IntegralHermiteGauss(_Ty& dValue);

//切比雪夫法求积
template <class _Ty>
_Ty IntegralChebyshev(_Ty a, _Ty b, _Ty eps);

//蒙特卡洛法求积
template <class _Ty >
_Ty IntegralMonteCarlo1D(_Ty a, _Ty b);

//二重变步长辛卜生法求积
template <class _Ty>
_Ty IntegralSimpson2D(_Ty a, _Ty b, _Ty eps);

//多重高斯法求积
template <class _Ty >
_Ty IntegralGaussMD(valarray<_Ty>& js);

//二重连分式法求积
template <class _Ty >
_Ty IntegralFraction2D(_Ty a, _Ty b, _Ty eps);

//多重蒙特卡洛法求积
template <class _Ty >
_Ty IntegralMonteCarlo2D(valarray<_Ty>& a, valarray<_Ty>& b);

# include "Integral.inl"  //类及相关函数的定义头文件

# endif    // _INTEGRAL_H
```

10.2 各种数值积分计算方法

10.2.1 变步长梯形法求积

1. 功能

用变步长梯形求积法计算定积分 $T=\int_a^b f(x)\mathrm{d}x$。

2. 算法

首先用梯形公式计算

$$T_n = \frac{h}{2}[f(a) + f(b)]$$

其中 $n = 1, h = b - a$ 。

然后用下列递推公式计算：

$$T_{2n} = \frac{1}{2}T_n + \frac{h}{2}\sum_{i=0}^{n-1} f(x_i + 0.5h)$$

$$2n \Rightarrow n, 0.5h \Rightarrow h$$

直至 $|T_{2n} - T_n| < \varepsilon$ 为止。

3. 函数原型

```
//变步长梯形法求积
template <class _Ty>
_Ty IntegralTrapezia(_Ty a, _Ty b, _Ty eps);
```

4. 参数

a：_Ty 类型变量。积分下限。

b：_Ty 类型变量。积分上限。要求 $b > a$ 。

eps：_Ty 类型变量。积分精度要求。

5. 返回值

_Ty 类型的积分值。

6. 函数说明

(1) 本函数需要调用计算被积函数 $f(x)$ 值的函数 FunctionValueIT()，需要由用户编写，其形式为：

```
template <class _Ty>
_Ty FunctionValueIT(_Ty dx)
{
   _Ty y;
   ……
   y = 被积函数 f(x)的表达式;
   return (y);
}
```

(2) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用变步长梯形求积法计算定积分

$$T = \int_0^1 e^{-x^2} dx$$

取 $\varepsilon = 0.000001$ 。

主函数程序及计算被积函数 $f(x)$值的函数示例程序如下：

```
//IntegralTrapezia.cpp        变步长梯形法求积

#include <iostream>           //输入输出流头文件
#include "Integral.h"         //积分头文件
```

```
using namespace std;          //名字空间

void main(void)
{
  double eps = FLOATERROR, a(0), b(1);

  cout << "IntegralTrapezia()" << endl << endl;

  double t = IntegralTrapezia(a,b,eps);

  cout << "t = " << t << endl << endl;
}

// 计算被积函数值
template <class _Ty>
_Ty FunctionValueIT(_Ty x)
{
  _Ty y;

  y = exp(-x*x);

   return(y);
}
```

8. 示例结果

```
IntegralTrapezia()

t = 0.746824
```

10.2.2 变步长辛卜生法求积

1. 功能

用变步长辛卜生(Simpson)求积法计算定积分 $S=\int_a^b f(x)\mathrm{d}x$ 。

2. 算法

变步长辛卜生求积法的步骤如下。

(1) 用梯形公式计算 $T_n=h[f(a)+f(b)]/2$,其中 $n=1,h=b-a$,且令 $S_n=T_n$ 。

(2) 用变步长梯形法则计算

$$T_{2n}=\frac{1}{2}T_n+\frac{h}{2}\sum_{k=0}^{n-1}f(x_k+h/2)$$

(3) 用辛卜生求积公式计算

$$S_{2n}=(4T_{2n}-T_n)/3$$

(4) 若 $|S_{2n}-S_n|\geqslant\varepsilon$,则令 $2n\Rightarrow n,h/2\Rightarrow h$,转(2)继续进行计算;否则结束。$S_{2n}$ 即为所求积分的近似值。其中 ε 为事先给定的求积精度。

3. 函数原型

```
//变步长辛卜生法求积
template <class _Ty>
_Ty IntegralSimpson1D(_Ty a, _Ty b, _Ty eps);
```

4. 参数

a:_Ty 类型变量。积分下限。

b:_Ty 类型变量。积分上限。要求 $b>a$。

eps:_Ty 类型变量。积分的精度要求。

5. 返回值

_Ty 类型的积分值。

6. 函数说明

(1) 本函数需要调用计算被积函数 $f(x)$ 值的函数 FunctionValueIS1D(),需要由用户编写,其形式为:

```
template <class _Ty>
_Ty FunctionValueIS1D(_Ty dx)
{
   _Ty y;
   ......
       y = 被积函数 f(x)的表达式;
   return (y);
}
```

(2) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用步长辛卜生求积法计算积分

$$S=\int_0^1 \frac{\ln(1+x)}{1+x^2}\mathrm{d}x$$

取 $\varepsilon = 0.000001$。

主函数程序及计算被积函数 $f(x)$ 值的函数示例程序如下:

```
//IntegralSimpson1D.cpp       变步长辛卜生法求积

#include <iostream>           //输入输出流头文件
#include "Integral.h"         //积分头文件
using namespace std;          //名字空间

void main(void)
{
  double eps = FLOATERROR, a(0), b(1);
  cout << "IntegralSimpson1D()" << endl << endl;
  double t = IntegralSimpson1D(a, b, eps);
  cout << "t = " << t << endl << endl;
}
```

```
// 计算被积函数值
template <class _Ty>
_Ty FunctionValueIS1D(_Ty x)
{
  _Ty y;
  y = log(1.0 + x) / (1.0 + x * x);
  return(y);
}
```

8. 示例结果

```
IntegralSimpson1D()

t = 0.272198
```

10.2.3 自适应梯形法求积

1. 功能

用自适应梯形求积法计算被积函数 $f(x)$ 在积分区间内有强峰的定积分 $T=\int_a^b f(x)\mathrm{d}x$。

2. 算法

自适应梯形求积法的过程如下。

首先将积分区间$[a,b]$分割成两个相等的子区间(称为 1 级子区间)$\Delta_0^{(1)}$ 和 $\Delta_1^{(1)}$,并在每一个子区间上分别用梯形公式计算积分近似值,设其结果分别为 $t_0^{(1)}$ 和 $t_1^{(1)}$。

然后将子区间 $\Delta_0^{(1)}$ 分割为两个相等的子区间(称为 2 级子区间)$\Delta_0^{(2)}$ 和 $\Delta_1^{(2)}$,并在每一个子区间上也分别用梯形公式计算积分近似值,设分别为 $t_0^{(2)}$ 和 $t_1^{(2)}$。如果下列不等式

$$|t_0^{(1)}-(t_0^{(2)}+t_1^{(2)})|<\varepsilon/1.4$$

成立,则保留 $t_0^{(2)}$ 和 $t_1^{(2)}$,再将 $\Delta_1^{(1)}$ 也分割为两个相等的 2 级子区间 $\Delta_2^{(2)}$ 和 $\Delta_3^{(2)}$,其相应的积分近似值为 $t_2^{(2)}$ 和 $t_3^{(2)}$。如果下列不等式

$$|t_1^{(1)}-(t_2^{(2)}+t_3^{(2)})|<\varepsilon/1.4$$

成立,则保留 $t_2^{(2)}$ 和 $t_3^{(2)}$。最后可得到满足精度要求的积分近似值为

$$t=t_0^{(2)}+t_1^{(2)}+t_2^{(2)}+t_3^{(2)}$$

如果在上述不等式中有一个不成立,则将对应的 2 级子区间再分割成两个相等的 3 级子区间。在考虑 3 级子区间时,其精度要求变为 $\varepsilon/1.4^2$。

同样,对 3 级子区间中不满足精度要求的子区间又可以分割成两个相等的 4 级子区间,其精度要求变为 $\varepsilon/1.4^3$。依此类推,这个过程一直进行到在所考虑的所有子区间上都满足精度要求为止。

本算法为递归算法。

3. 函数原型

```
//自适应梯形法求积
template <class _Ty>
_Ty IntegralTrapeziaSelfAdapt(_Ty a, _Ty b, _Ty eps, _Ty d);
```

4. 参数

a:_Ty 类型变量。积分下限。

b:_Ty 类型变量。积分上限。要求 $b>a$。

eps:_Ty 类型变量。积分精度要求。

d:_Ty 类型变量。对积分区间进行分割的最小步长，当子区间的宽度小于 d 时，即没有满足精度要求，也不再往下进行分割。

5. 返回值

_Ty 类型的积分值。

6. 函数说明

(1) 本函数需要调用计算被积函数 $f(x)$ 值的函数 FunctionValueITSA()，需要由用户编写，其形式为：

```
template <class _Ty>
_Ty FunctionValueITSA(_Ty dx)
{
   _Ty y;
   ……
   y = 被积函数 f(x)的表达式;
   return (y);
}
```

(2) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用自适应梯形求积分法计算定积分

$$T=\int_{-1}^{1}\frac{1}{1+25x^2}\mathrm{d}x$$

取 $\varepsilon=0.000001$，最小步长为 0.0001。

主函数程序及计算被积函数 $f(x)$ 值的函数示例程序如下：

```
//IntegralTrapeziaSelfAdapt.cpp          自适应梯形法求积

#include <iostream>                      //输入输出流头文件
#include "Integral.h"                    //积分头文件
using namespace std;                     //名字空间

void main(void)
{
  double eps = FLOATERROR, a(-1), b(1), d(0.0001);
  cout << "IntegralTrapeziaSelfAdapt()" << endl << endl;
  double dIntegralValue = IntegralTrapeziaSelfAdapt(a,b,eps,d);
  cout << "IntegralValue = " << dIntegralValue << endl << endl;
}

// 计算被积函数值
template <class _Ty>
_Ty FunctionValueITSA(_Ty x)
{
   _Ty y;
```

```
    y = 1.0 / (1.0 + 25.0 * x * x);
    return(y);
}
```

8. 示例结果

```
IntegralTrapeziaSelfAdapt()

IntegralValue = 0.549363
```

10.2.4 龙贝格法求积

1. 功能

用龙贝格(Romberg)求积法计算定积分 $T=\int_a^b f(x)\mathrm{d}x$ 。

2. 算法

高 $T_m(h)$为步长为 h 时利用 $2m-2$ 阶牛顿－阿特斯(Newton-Cotes)公式计算得到的结果；$T_m\left(\frac{h}{2}\right)$为将步长 h 减半后用 $2m-2$ 阶牛顿－阿特斯公式计算得到的结果。将它们进行线性组合，得到步长为 h 的 $2m$ 阶牛顿－柯特斯公式，即

$$T_{m+1}(h)=\frac{4^m T_m(h/2)-T_m(h)}{4^m-1}$$

其中 $T_1(h)$为步长为 h 时的梯形公式计算得到的结果，$T_1(h/2)$为步长为 $h/2$ 时的梯形公式计算得到的结果。

在实际进行计算时，龙贝格求积法按下表所示的计算格式进行，直到 $|T_{m+1}(h)-T_m(h)|<\varepsilon$ 为止。

梯形法则	二阶公式	四阶公式	六阶公式	八阶公式	…
$T_1(h)$					
$T_1\left(\frac{h}{2}\right)$	$T_2(h)$				
$T_1\left(\frac{h}{2^2}\right)$	$T_2\left(\frac{h}{2}\right)$	$T_3(h)$			
$T_1\left(\frac{h}{2^3}\right)$	$T_2\left(\frac{h}{2^2}\right)$	$T_3\left(\frac{h}{2}\right)$	$T_4(h)$		
$T_1\left(\frac{h}{2^4}\right)$	$T_2\left(\frac{h}{2^3}\right)$	$T_3\left(\frac{h}{2^2}\right)$	$T_4\left(\frac{h}{2}\right)$	$T_5(h)$	…
⋮	⋮	⋮	⋮	⋮	

在本函数中，最多可以算到 $m=10$，如果此时还达不到精度要求，就取 T_{10}作为最后结果。

3. 函数原型

```
//龙贝格法求积
template <class _Ty>
_Ty IntegralRomberg(_Ty a, _Ty b, _Ty eps);
```

4. 参数

a：_Ty 类型变量。积分下限。

b：_Ty 类型变量。积分上限。要求 $b>a$。

eps：_Ty 类型变量。积分精度要求。

5. 返回值

_Ty 类型的积分值。

6. 函数说明

(1) 本函数需要调用计算被积函数 $f(x)$值的函数 FunctionValueR(),需要由用户编写,其形式为:

```
template <class _Ty>
_Ty FunctionValueR(_Ty dx)
{
   _Ty y;
   ……
   y = 被积函数 f(x)的表达式;
   return (y);
}
```

(2) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用龙贝格求积法计算定积分

$$T=\int_0^1 \frac{x}{4+x^2}\mathrm{d}x$$

取 ε=0.000001。

主函数程序及计算被积函数 $f(x)$值的函数示例程序如下:

```
//IntegralRomberg.cpp        龙贝格法求积

#include <iostream>          //输入输出流头文件
#include "Integral.h"        //积分头文件
using namespace std;         //名字空间

void main(void)
{
  double eps = FLOATERROR, a(0), b(1);
  cout << "IntegralRomberg()" << endl << endl;
  double dIntegralValue = IntegralRomberg(a,b,eps);
  cout << "IntegralValue = " << dIntegralValue << endl << endl;
}

// 计算被积函数值
template <class _Ty>
_Ty FunctionValueR(_Ty x)
{
   _Ty y;
   y = x / (4.0 + x * x);
   return(y);
}
```

8. 示例结果

```
IntegralRomberg()

IntegralValue = 0.111572
```

10.2.5 一维连分式法求积

1. 功能

用连分式求积法计算定积分 $s=\int_a^b f(x)\mathrm{d}x$ 。

2. 算法

利用变步长梯形法则计算出步长分别为

$$h_i=(b-a)/2^i, i=0,1,\cdots$$

的一系列积分近似值 $s_i(i=0,1,\cdots)$。显然，积分近似值 s 是步长 h 的函数 $s(h)$。

将函数 $s(h)$用如下连分式表示

$$s(h)=b_0+\frac{h-h_0}{b_1}+\frac{h-h_1}{b_2}+\cdots+\frac{h-h_{i-1}}{b_i}+\cdots$$

其中 $b_0,b_1,\cdots,b_i,\cdots$可以由一系列的积分近似值点$(h_i,s_i)(i=0,1,\cdots)$来确定，其计算公式如下：

$$\begin{aligned}
&b_0=s_0\\
&b_1=\frac{h_1-h_0}{s_1-b_0}\\
&b_{21}=\frac{h_2-h_0}{s_2-b_0}, b_2=\frac{h_2-h_1}{b_2-b_1}\\
&b_{31}=\frac{h_3-h_0}{s_3-b_0}, b_{32}=\frac{h_3-h_1}{b_{31}-b_1}, b_3=\frac{h_3-h_2}{b_{32}-b_2}\\
&\quad\vdots
\end{aligned}$$

一般说来，如果已知 $b_0,b_1,\cdots,b_{i-1}$，则在计算出一个新的积分近似值点(h_i,s_i)后，可以用以下递进公式计算 b_i：

$$\begin{cases}
b_{i1}=s_i\\
b_{ij}=\dfrac{h_i-h_{j-1}}{b_{i,j-1}-b_{j-1}}, j=1,2,\cdots,i\\
b_i=b_{ii}
\end{cases}$$

当 $h=0$ 时，$s(0)$即为积分的精确值，即

$$s=s(0)=b_0-\frac{h_0}{b_1}-\frac{h_1}{b_2}-\cdots-\frac{h_{i-1}}{b_i}-\cdots$$

在实际进行计算时，一般取到七节就能满足精度要求。在本函数中，最多可以取到十节，如果此时还不满足精度要求，就直接取当前的积分近似值。

3. 函数原型

```
//一维连分式法求积
template <class _Ty>
_Ty IntegralFraction1D(_Ty a, _Ty b, _Ty eps);
```

4. 参数

a:_Ty 类型变量。积分下限。

b:_Ty 类型变量。积分上限。$b>a$。

eps:_Ty 类型变量。积分精度要求。

5. 返回值

_Ty 类型的积分值。

6. 函数说明

(1) 本函数需要调用计算被积函数 $f(x)$值的函数 FunctionValueF1D(),需要由用户编写,其形式为:

```
template <class _Ty>
_Ty FunctionValueF1D(_Ty dx)
{
   _Ty y;
   ......
   y = 被积函数 f(x)的表达式;
   return (y);
}
```

(2) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用连分式法计算定积分

$$s=\int_0^{4.3} e^{-x^2}dx$$

取 $\varepsilon=0.000001$。

主函数程序及计算被积函数 $f(x)$值的函数示例程序如下:

```
//IntegralFraction1D.cpp        一维连分式法求积

#include <iostream>             //输入输出流头文件
#include "Integral.h"           //积分头文件
using namespace std;            //名字空间

void main(void)
{
  double eps = FLOATERROR, a(0), b(4.3);
  cout << "IntegralFraction1D()" << endl << endl;
  double dIntegralValue = IntegralFraction1D(a,b,eps);
  cout << "IntegralValue = " << dIntegralValue << endl << endl;
}

// 计算被积函数值
template <class _Ty>
_Ty FunctionValueF1D(_Ty x)
{
```

```
    _Ty y;
    y = exp(-x * x);
    return(y);
}
```

8. 示例结果

```
IntegralFraction1D()

IntegralValue = 0.886227
```

10.2.6 高振荡函数法求积

1. 功能

用分部积分法计算高振荡函数的积分

$$\int_a^b f(x)\sin mx\,\mathrm{d}x \text{ 与} \int_a^b f(x)\cos mx\,\mathrm{d}x$$

其中 m 比较大。

2. 算法

当 m 较大时,积分

$$s_1(m) = \int_a^b f(x)\cos mx\,\mathrm{d}x$$

与

$$s_2(m) = \int_a^b f(x)\sin mx\,\mathrm{d}x$$

称为高振荡积分。

令

$$s(m) = \int_a^b f(x)\mathrm{e}^{jmx}\,\mathrm{d}x$$

其中 $\mathrm{e}^{jmx} = \cos mx + j\sin mx$,则有

$$s(m) = s_1(m) + js_2(m)$$

反复利用分部积分法可以得到

$$s(m)=\int_a^b f(x)\mathrm{e}^{jmx}\,\mathrm{d}x$$

$$\approx -\sum_{k=0}^{n-1}\left(\frac{j}{m}\right)^{k+1}\left[\left(f^{(k)}(b)\cos mb - f^{(k)}(a)\cos ma\right)+j\left(f^{(k)}(b)\sin mb - f^{(k)}(a)\sin ma\right)\right]$$

分离出实部和虚部后就得到

$$s_1(m)=\int_a^b f(x)\cos mx\,\mathrm{d}x$$

$$\approx \sum_{k=0}^{n-1}\frac{1}{m^{k+1}}\left[f^{(k)}(b)\sin\left(\frac{k\pi}{2}+mb\right)-f^{(k)}(a)\sin\left(\frac{k\pi}{2}+ma\right)\right]$$

$$s_2(m)=\int_a^b f(x)\sin mx\,\mathrm{d}x$$

$$\approx \sum_{k=0}^{n-1}\frac{-1}{m^{k+1}}\left[f^{(k)}(b)\cos\left(\frac{k\pi}{2}+mb\right)-f^{(k)}(a)\cos\left(\frac{k\pi}{2}+ma\right)\right]$$

其中

$$\sin\left(\frac{k\pi}{2}+\pi\right)=\begin{cases}\sin x, \mathrm{mod}(k,4)=0\\ \cos x, \mathrm{mod}(k,4)=1\\ -\sin x, \mathrm{mod}(k,4)=2\\ -\cos x, \mathrm{mod}(k,4)=3\end{cases}$$

$$\cos\left(\frac{k\pi}{2}+\pi\right)=\begin{cases}\cos x, \mathrm{mod}(k,4)=0\\ -\sin x, \mathrm{mod}(k,4)=1\\ -\cos x, \mathrm{mod}(k,4)=2\\ \sin x, \mathrm{mod}(k,4)=3\end{cases}$$

n 的大小可根据 $f(x)$ 及 m 的大小而定，一般取 $n=3$。

3. 函数原型

```
//高振荡函数法求积
template <class _Ty>
void IntegralSurge(_Ty a, _Ty b, int m, valarray<_Ty>& fa,
                   valarray<_Ty>& fb, valarray<_Ty>& s);
```

4. 参数

a：_Ty 类型变量。积分下限。

b：_Ty 类型变量。积分上限。要求 b>a。

m：整形变量。被积函数中振荡函数的角频率。

fa：_Ty 类型一维组数，长度为 n，存放在 $f(x)$ 在积分区间端点 $x=a$ 处的各阶导数值，即

$$fa(i)=f^{(i)}(a), i=0,1,\cdots,n-1$$

fb：_Ty 类型一维组数，长度为 n，存放在 $f(x)$ 在积分区间端点 $x=b$ 处的各阶导数值，即

$$fb(i)=f^{(i)}(b), i=0,1,\cdots,n-1$$

s：_Ty 类型一维组数，长度为 2，其中 $s(0)$ 返回积分

$$\int_a^b f(x)\cos mx\,\mathrm{d}x$$

的值；$s(1)$ 返回积分

$$\int_a^b f(x)\sin mx\,\mathrm{d}x$$

的值。

5. 返回值

_Ty 类型的积分值。

6. 函数说明

所有输入和输出的数据都应使用浮点类型。

7. 示例

用分部积分法计算下列高振荡积分

$$s_1=\int_0^{2\pi} x\cos x\cos 30x\,\mathrm{d}x$$

与

$$s_2=\int_0^{2\pi} x\cos x\sin 30x\,\mathrm{d}x$$

其中 $a=0.0, b=6.2831852, m=30$。

取 $n=4$　　$f(x)=x\cos x$，　　$f'(x)=\cos x-x\sin x$

$f''(x)=-2\sin x-x\cos x$，　　$f'''(x)=-3\cos x+x\sin x$

则有

$fa(0)=f^{(0)}(0)=0.0$，　　$fa(1)=f^{(1)}(0)=1.0$

$fa(2)=f^{(2)}(0)=0.0$，　　$fa(3)=f^{(3)}(0)=-3.0$

$fb(0)=f^{(0)}(2\pi)=6.2831852$，　　$fb(1)=f^{(1)}(2\pi)=1.0$

$fb(2)=f^{(2)}(0)=-6.2831852$，　　$fb(3)=f^{(3)}(2\pi)=-3.0$

主函数示例程序如下：

```
//IntegralSurge.cpp        高振荡函数法求积

#include <iostream>        //输入输出流头文件
#include "Integral.h"      //积分头文件
using namespace std;       //名字空间

void main(void)
{
  double eps = FLOATERROR, a(0), b(6.2831852);
  double fa[4] = {0, 1, 0, -3};
  double fb[4] = {6.2831852, 1, -6.2831852, -3};
  int m(30);
  valarray<double> ffa(fa, 4);
  valarray<double> ffb(fb, 4);
  valarray<double> s(2);

  cout << "IntegralSurge()" << endl << endl;

  IntegralSurge(a, b, m, ffa, ffb, s);

  cout << "s(0) = " << s[0] << "\t s(1) = " << s[1] << endl << endl;
}
```

8. 示例结果

```
IntegralSurge()

s(0) =  -6.74177e-007 s(1) =  -0.209672
```

10.2.7　勒让德—高斯法求积

1. 功能

用勒让德—高斯(Legendre-Gauss)求积法计算定积分 $G=\int_a^b f(x)\mathrm{d}x$。

2. 算法

对积分变量 x 作变换 $x=\frac{b-a}{2}t+\frac{b+a}{2}$。

将原积分化为在区间[−1,1]上的积分，即

$$G=\int_a^b f(x)\mathrm{d}x=\frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}t+\frac{b+a}{2}\right)\mathrm{d}t$$

$$=\frac{b-a}{2}\int_{-1}^{1}\varphi(t)\mathrm{d}t$$

根据插值求积公式有

$$\int_{-1}^{1}\varphi(t)\mathrm{d}t=\sum_{k=0}^{n-1}\lambda_k\varphi(t_k)$$

其中 $t_k(k=0,1,\cdots,n-1)$ 为在区间[−1,1]上的 n 个求积结点，且

$$\lambda_k=\int_{-1}^{1}A_k(t)\mathrm{d}t$$

$$A_k(t)=\prod_{\substack{j=0\\ j\neq k}}^{n-1}[(t-t_j)/(t_k-t_j)]$$

如果 n 个求积结点 $t_k(k=0,1,\cdots,n-1)$ 取勒让德(Legendre)多项式

$$\frac{1}{2^n n!}\frac{d^n}{dt^n}[(t^2-1)^n]$$

在区间[−1,1]上的 n 个零点，由上述插值求积公式称为勒让德−高斯求积公式，其代数精确度为 $2n-1$。

下表列出了 n 从 2 到 10 之间的求积结点 t_k 和求积系数 λ_k。

n	求积结点 t_k	求积系数 λ_k
2	±0.5773502692	1.0
3	0.0 ±0.7745966692	0.8888888889 0.5555555556
4	±0.3399810436 ±0.8611363116	0.6521451549 0.3478548451
5	0.0 ±0.5384693101 ±0.9061798459	0.5688888889 0.4786286705 0.2369268851
6	±0.2386191861 ±0.6612093865 ±0.9324695142	0.4679139346 0.3607615731 0.1713244924
7	0.0 ±0.4058451514 ±0.7415311856 ±0.9491079123	0.4179591837 0.3818300505 0.2797053915 0.1294849662
8	±0.1834346425 ±0.5255324099 ±0.7966664774 ±0.9602898656	0.3626837834 0.3137066459 0.2223801345 0.1012285363
9	0.0 ±0.3242534234 ±0.6133714327 ±0.8360311073 ±0.9681602395	0.3302393550 0.3123470770 0.2606106964 0.1806481607 0.0812743884
10	±0.1488743390 ±0.4333953941 ±0.6794095683 ±0.8650633667 ±0.9739065285	0.2955242247 0.2692667193 0.2190863625 0.1494513492 0.0666713443

在本函数中，取 $n=5$，并采取变步长求积法。

3. 函数原型

```
//勒让德—高斯法求积
template <class _Ty>
_Ty IntegralLegendreGauss(_Ty a, _Ty b, _Ty eps);
```

4. 参数

a：_Ty 类型变量。积分下限。

b：_Ty 类型变量。积分上限。要求 $b>a$。

eps：_Ty 类型变量。积分的精度要求。

5. 返回值

_Ty 类型的积分值。

6. 函数说明

(1) 本函数需要调用计算被积函数 $f(x)$ 值的函数 FunctionValueLegG()，需要由用户编写，其形式为：

```
template <class _Ty>
_Ty FunctionValueLegG(_Ty dx)
{
    _Ty y;
    ……
    y = 被积函数 f(x)的表达式;
    return (y);
}
```

(2) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用勒让得—高斯求积法计算定积分

$$G=\int_{2.5}^{8.4}(x^2+\sin x)\mathrm{d}x$$

取 $\varepsilon=0.000001$。

主函数程序及计算被积函数 $f(x)$ 的函数示例程序如下：

```
//IntegralLegendreGauss.cpp  勒让德—高斯法求积

#include <iostream>          //输入输出流头文件
#include "Integral.h"        //积分头文件
using namespace std;         //名字空间

void main(void)
{
  double eps = FLOATERROR, a(2.5), b(8.4);
  double dValue = IntegralLegendreGauss(a, b, eps);
  cout << "IntegralLegendreGauss()" << endl << endl;
  cout << "dValue = " << dValue << endl << endl;
}
```

```
// 计算被积函数值
template <class _Ty>
_Ty FunctionValueLegG(_Ty x)
{
  _Ty dValue;
  dValue = x * x + sin(x);
  return(dValue);
}
```

8. 示例结果

```
IntegralLegendreGauss()

dValue = 192.078
```

10.2.8 拉盖尔—高斯法求积

1. 功能

用拉盖尔—高斯(Laguerre-Gauss)求积公式计算半无限区间[0,∞)上的积分

$$G=\int_0^{\infty} f(x)\mathrm{d}x$$

2. 算法

设半区间[0,∞)上的积分为

$$G=\int_0^{\infty} f(x)\mathrm{d}x$$

n 点拉盖尔—高斯求积公式为

$$G=\sum_{i=0}^{n-1}\lambda_i f(x_i)$$

其中 $x_i(i=0,1,\cdots,n-1)$ 为 n 阶拉盖尔多项式

$$L_n(x)=\mathrm{e}^x\frac{\mathrm{d}^n}{\mathrm{d}x^n}(x^n\mathrm{e}^{-x}),0\leqslant x<+\infty$$

的 n 个零点，λ_i 为求积系数。

在本函数中，取 $n=5$，5 阶拉盖尔—高斯求积公式的结点为

$x_0=0.26355990$，　$x_1=1.41340290$，　$x_2=3.59642600$

$x_3=7.08580990$，　$x_4=12.64080000$

求积系数为

$\lambda_0=0.6790941054$，　$\lambda_1=1.638487956$，　$\lambda_2=2.769426772$

$\lambda_3=4.315944000$，　$\lambda_4=7.104896230$

本方法特别适用于计算如下形式的积分：

$$\int_0^{\infty}\mathrm{e}^{-x}g(x)\mathrm{d}x$$

3. 函数原型

```
//拉盖尔—高斯法求积
template <class _Ty>
void IntegralLaguerreGauss(_Ty& dValue);
```

4. 参数

dValue：_Ty 类型引用，程序结束时返回积分值。

5. 返回值

无。

6. 函数说明

(1) 本函数需要调用计算被积函数 $f(x)$ 值的函数 FunctionValueLagG()，需要由用户编写，其形式为：

```
long double FunctionValueLagG(long double dx)
{
   long double y;
   ……
   y = 被积函数 f(x)的表达式;
   return (y);
}
```

(2) 输入的数据应使用浮点类型，最好为 long double 类型。

7. 示例

计算半无限区间的积分

$$G = \int_0^{\infty} x e^{-x} dx$$

主程序及计算被积函数 $f(x)$ 值的函数示例程序如下：

```
//IntegralLaguerreGauss.cpp  拉盖尔—高斯法求积

#include <iostream>          //输入输出流头文件
#include "Integral.h"        //积分头文件
using namespace std;         //名字空间

void main(void)
{
  long double dValude;
  IntegralLaguerreGauss(dValude);
  cout << "IntegralLaguerreGauss()" << endl << endl;
  cout << "Integral Valude = " << dValude << endl;
}

// 计算被积函数值
template <class _Ty>
_Ty FunctionValueLagG(_Ty x)
{
_Ty y;
y = x * exp(-x);
   return(y);
}
```

8. 示例结果

```
IntegralLaguerreGauss()

Integral Valude = 0.999995
```

10.2.9 埃尔米特—高斯法求积

1. 功能

用埃尔米特—高斯(Hernite-Gauss)求积公式计算无限区间$(-\infty,\infty)$上的积分

$$G=\int_{-\infty}^{\infty}f(x)\mathrm{d}x$$

2. 算法

设无限区间$(-\infty,\infty)$上的积为

$$G=\int_{-\infty}^{\infty}f(x)\mathrm{d}x$$

n点埃尔米特—高斯求积公式为

$$G=\sum_{i=0}^{n-1}\lambda_i f(x_i)$$

其中$x_i(i=0,1,\cdots,n-1)$为n阶埃尔米特多项式

$$H_n(x)=(-1)^n\mathrm{e}^{x^2}\frac{\mathrm{d}^n}{\mathrm{d}x^n}(\mathrm{e}^{x^{-2}}),\quad -\infty<x<+\infty$$

的n个零点,λ_i为求积系数。

在本函数中,取$n=5$,5阶埃尔米特—高斯求积公式的结点为

$$x_0=-2.02018200,\quad x_1=-0.95857190,\quad x_2=0.0$$
$$x_3=0.95857190,\quad x_4=2.02018200$$

求积系数为

$$\lambda_0=1.181469599,\quad \lambda_1=0.9865791417,\quad \lambda_2=0.9453089237$$
$$\lambda_3=0.9865791417,\quad \lambda_4=1.181469599$$

本方法特别适用于计算如下形式的积分:

$$\int_{-\infty}^{\infty}\mathrm{e}^{-x^2}g(x)\mathrm{d}x$$

3. 函数原型

```
//埃尔米特—高斯法求积
template <class _Ty>
void IntegralHermiteGauss(_Ty& dValue);
```

4. 参数

dValue:_Ty类型引用,程序结束时返回积分值。

5. 返回值

无。

6. 函数说明

(1) 本函数需要调用计算被积函数$f(x)$值的函数FunctionValueHG(),需要由用户编写,其形式为:

```
template <class _Ty>
```

```
_Ty FunctionValueHG(_Ty dx)
{
    _Ty y;
    ……
    y = 被积函数 f(x)的表达式;
    return (y);
}
```

(2) 输入的数据应使用浮点类型,最好为 long double 类型。

7. 示例

计算无限区间积分

$$G=\int_{-\infty}^{\infty}x^2e^{-x^2}dx$$

主函数程序及计算被积函数 $f(x)$值的函数示例程序如下:

```
//IntegralHermiteGauss.cpp  埃尔米特—高斯法求积

#include <iostream>          //输入输出流头文件
#include "Integral.h"        //积分头文件
using namespace std;         //名字空间

void main(void)
{
  long double dValue;
  cout << "IntegralHermiteGauss()" << endl << endl;
  IntegralHermiteGauss(dValue);
  cout << "dValue = " << dValue << endl << endl;
}

// 计算被积函数值
template <class _Ty>
_Ty FunctionValueHG(_Ty x)
{
    _Ty dValue;
    dValue = x * x * exp(-x * x);
    return(dValue);
}
```

8. 示例结果

```
IntegralHermiteGauss()

dValue = 0.886223
```

10.2.10 切比雪夫法求积

1. 功能

用切比雪夫(Chebyshev)求积公式计算定积分 $S=\int_a^b f(x)dx$。

2. 算法

对积分变量 x 作变换

$$x=\frac{b-a}{2}t+\frac{b+a}{2}$$

将原积分化为在区间[-1,1]上的积分

$$s=\frac{b-a}{2}\int_{-1}^{1}f\left(\frac{b-a}{2}t+\frac{b+a}{2}\right)\mathrm{d}t=\frac{b-a}{2}\int_{-1}^{1}\varphi(t)\mathrm{d}t$$

切比雪夫求积公式为

$$\int_{-1}^{1}\varphi(t)\mathrm{d}t=\frac{2}{n}\sum_{k=0}^{n-1}\varphi(t_k)$$

当 $n=5$ 时,有

$$t_0=-0.8324975,\quad t_1=-0.3745414$$
$$t_2=0.0,\quad t_3=0.3745414$$
$$t_4=0.8324975$$

3. 函数原型

```
//切比雪夫法求积
template <class _Ty>
_Ty IntegralChebyshev(_Ty a, _Ty b, _Ty eps);
```

4. 参数

a:_Ty 类型变量。积分下限。

b:_Ty 类型变量。积分上限。要求 $b>a$。

eps:_Ty 类型变量。积分精度要求。

5. 返回值

_Ty 类型的积分值。

6. 函数说明

(1) 本函数需要调用计算被积函数 $f(x)$ 值的函数 FunctionValueIS1D(),该函数需要由用户编写,其形式为:

```
template <class _Ty>
_Ty FunctionValueCb(_Ty dx)
{
  _Ty y;
  ……
  y = 被积函数 f(x)的表达式;
  return (y);
}
```

(2) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用切比雪夫求积法计算定积分

$$S=\int_{2.5}^{8.4}(x^2+\sin x)\mathrm{d}x$$

取 $\varepsilon=0.000001$。

主函数程序及计算被积函数 $f(x)$ 值的函数示例程序如下:

```
//IntegralChebyshev.cpp  切比雪夫法求积

#include <iostream>          //输入输出流头文件
#include "Integral.h"        //积分头文件
using namespace std;         //名字空间

void main(void)
{
  double eps = FLOATERROR, a(2.5), b(8.4);
  cout << "IntegralChebyshev()" << endl << endl;
  double dValue = IntegralChebyshev(a, b, eps);
  cout << "dValue = " << dValue << endl << endl;
}

// 计算被积函数值
template <class _Ty>
_Ty FunctionValueCb(_Ty x)
{
  _Ty dValue;
  dValue = x * x + sin(x);
  return(dValue);
}
```

8. 示例结果

```
IntegralChebyshev()

dValue = 192.078
```

10.2.11 蒙特卡洛法求积

1. 功能

用蒙特卡洛(Monte Carlo)计算定积分。

2. 算法

设定积分为

$$S=\int_a^b f(x)\mathrm{d}x$$

取 0 到 1 之间均匀分布的随机数序列 $\gamma_i(i=0,1,\cdots,m-1)$,并令

$$x_i=a+(b-a)\gamma_i,\quad i=0,1,\cdots,m-1$$

只要 m 足够大,则有

$$S\approx\frac{b-a}{m}\sum_{i=0}^{m-1}f(x_i)$$

本函数取 $m=10000$。

3. 函数原型

```
//用蒙特卡洛法求积
template <class _Ty >
```

```
_Ty IntegralMonteCarlo1D(_Ty a, _Ty b);
```

4. 参数

a:_Ty 类型变量。积分下限。

b:_Ty 类型变量。积分上限。

5. 返回值

_Ty 类型的积分值。

6. 函数说明

(1) 本函数需要调用计算被积函数 $f(x)$值的函数 FunctionValueMC1D(),需要由用户编写,其形式为:

```
template <class _Ty>
_Ty FunctionValueMC1D(_Ty dx)
{
    _Ty y;
    ......
    y = 被积函数 f(x)的表达式;
    return (y);
}
```

(2) 所有输入和输出的数据都应使用浮点类型,最好为 double 型。

(3) 本函数调用了随机数的 rand_01_One()函数,请参阅第 3 章 3.2.2 节。

7. 示例

用蒙特卡洛法计算定积分

$$S = \int_{2.5}^{8.4} (x^2 + \sin x)\mathrm{d}x$$

主函数程序及计算被积函数 $f(x)$值的函数示例程序如下:

```
//IntegralMonteCarlo1D.cpp  用蒙特卡洛法求积

#include <iostream>          //输入输出流头文件
#include "Integral.h"        //积分头文件
using namespace std;         //名字空间

void main(void)
{
  double a(2.5), b(8.4);
  double dValue = IntegralMonteCarlo1D(a, b);   //调用求积函数
  cout << "IntegralMonteCarlo1D()" << endl << endl;
  cout << "dValue = " << dValue << endl << endl;
}

//计算被积函数值
template <class _Ty>
_Ty FunctionValueMC1D(_Ty x)
{
  _Ty dValue;
```

```
    dValue = x * x + sin(x);
    return(dValue);
}
```

8. 示例结果

```
IntegralMonteCarlo1D()

dValue = 191.553
```

10.2.12 二重变步长辛卜生法求积

1. 功能

用变步长辛卜生(Simpson)方法计算二重积分

$$S=\int_a^b \mathrm{d}x\int_{y_0(x)}^{y_1(x)} f(x,y)\mathrm{d}y$$

2. 算法

首先将二重积分化为两个单积分，即

$$g(x)=\int_{y_0(x)}^{y_1(x)} f(x,y)\mathrm{d}y \text{ 与 } s=\int_a^b g(x)\mathrm{d}x$$

然后对每一个单积分采用变步长辛卜生法则。其计算步骤如下。

(1) 固定一个 x，设为 $\bar{x}$。

① 用梯形公式计算

$$t_1=[y_1(\bar{x})-y_0(\bar{x})][f(\bar{x},y_0(\bar{x}))+f(\bar{x},y_1(\bar{x}))]/2$$

② 将区间分半，每一个子区间长度为

$$h_k=[y_1(\bar{x})-y_0(\bar{x})]/2^k,\quad k=1,2,\cdots$$

用辛卜生公式计算

$$t_{k+1}=\frac{1}{2}t_k+h_k\sum_{i=1}^{n} f(\bar{x},y_0(\bar{x})+(2i-1)h_k)$$

$$g_k=(4t_{k+1}-t_k)/3$$

其中 $n=2^{k-1}$

重复②，直至 $|g_k-g_{k-1}|<\varepsilon(1+|g_k|)$ 为止，此时即有 $g(\bar{x})=g_k$。

(2) 利用 1. 中所得到的一系列 $g(\bar{x})$ 值计算二重积分的近似值 s。

① 利用梯形公式计算 $u_1=(b-a)(g(b)+g(a))/2$。

② 将区间二等分，其子区间长度为 $h'_k=(b-a)/2^k, k=1,2,\cdots$

用辛卜生公式计算

$$u_{k+1}=\frac{1}{2}u_k+h'_k\sum_{i=1}^{n} g(a+h'_k(2i-1))$$

$$s_k=(4u_{k+1}-u_k)/3$$

其中 $n=2^{k-1}$。

重复②，直至 $|s_k-s_{k-1}|<\varepsilon(1+|s_k|)$ 为止，此时即有 $s\approx s_k$。

3. 函数原型

```
//二重变步长辛卜生法求积
```

```
template <class _Ty>
_Ty IntegralSimpson2D(_Ty a, _Ty b, _Ty eps);
```

4. 参数

a:_Ty 类型变量。对单变量 x 的积分下限。

b:_Ty 类型变量。对单变量 x 的积分上限。要求 $b>a$。

eps:_Ty 类型变量。积分精度要求。

5. 返回值

_Ty 类型积分值。

6. 函数说明

(1) 本函数需要调用计算上、下限 $y_1(x)$、$y_0(x)$(要求 $y_1(x)>y_0(x)$)的函数 FunctionBoundaryIS2D(),由用户自己编写,其形式为:

```
template <class _Ty>
void FunctionBoundaryIS2D(_Ty x, valarray<_Ty>& y)
{
  y[0] = 下限 y0(x)的表达式;
  y[1] = 上限 y1(x)的表达式;
}
```

(2) 本函数还要调用计算被积函数 $f(x,y)$值的函数 FunctionValueIS2D(),由用户自己编写,其形式为:

```
template <class _Ty>
_Ty FunctionValueIS2D(_Ty x, _Ty y)
{
      double z;
      z = 被积函数 f(x,y)的表达式;
      return(z);
}
```

(3) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用变步长辛卜生求积法计算二重积分

$$S=\int_0^1 \mathrm{d}x\int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} \mathrm{e}^{x^2+y^2}\,\mathrm{d}y$$

取 $\varepsilon=0.0001$。

主函数程序及计算上、下限值 $y_1(x)$、$y_0(x)$的函数程序与计算被积函数 $f(x,y)$值的函数示例程序如下:

```
//IntegralSimpson2D.cpp  二重变步长辛卜生法求积

#include <iostream>          //输入输出流头文件
#include "Integral.h"        //积分头文件
using namespace std;         //名字空间

void main(void)
{
```

```
  double eps = FLOATERROR, a(0), b(1);
  cout << "IntegralSimpson2D()" << endl << endl;
  double dValue = IntegralSimpson2D(a,b,eps);
  cout << "dValue = " << dValue << endl << endl;
}

// 计算上、下限 y1(x)、y0(x)值
template <class _Ty>
void FunctionBoundaryIS2D(_Ty x, valarray<_Ty>& y)
{
y[0] = -sqrt(1.0 - x * x);
   y[1] = -y[0];
}

// 计算被积 f(x,y)函数值
template <class _Ty>
_Ty FunctionValueIS2D(_Ty x, _Ty y)
{
  _Ty zValue;
  zValue = exp(x * x + y * y);
  return(zValue);
}
```

8. 示例结果

```
IntegralSimpson2D()

dValue = 2.69907
```

10.2.13 多重高斯法求积

1. 功能

用高斯(Gauss)方法计算 n 重积分

$$S=\int_{c_0}^{d_0}\mathrm{d}x_0\int_{c_1(x_0)}^{d_1(x_0)}\mathrm{d}x_1\int_{c_1(x_0,x_1)}^{d_2(x_0,x_1)}\mathrm{d}x_2\cdots\int_{c_{n-1}(x_0,x_1,\cdots,x_{n-2})}^{d_{n-1}(x_0,x_1,\cdots,x_{n-2})}f(x_0,x_1,\cdots,x_{n-1})\mathrm{d}x_{n-1}$$

2. 算法

在计算 n 重积分时，分别将 $0,1,\cdots,n-1$ 层区间分为各自相等的 $js_0,js_1,\cdots,js_{n-1}$ 个子区间。首先求出各层积分区间上的第一个子区间中第一个高斯型点 $\overline{x}_0,\overline{x}_1,\cdots,\overline{x}_{n-1}$；然后固定 $\overline{x}_0,\overline{x}_1,\cdots,\overline{x}_{n-2}$；按高斯方法计算最内层(即第 $n-1$ 层)的积分，再从内到外计算各层积分值。最后就得到所要求的 n 重积分的近似值。

计算单重积分的高斯方法见本章 10.2.7 节的算法说明。

在本函数中，每个子区间上取 5 个高斯型结点。

3. 函数原型

```
//多重高斯法求积
template <class _Ty >
_Ty IntegralGaussMD(valarray<_Ty>& js);
```

4. 参数

js:整形一维组数,长度为 n。$js(i)(i=0,1,\cdots,n-1)$表示第 i 层积分区间所划分的子区间个数。

5. 返回值

_Ty 类型积分值。

6. 函数说明

(1) 本函数需要调用计算各层积分上、下限(要求所有的上限大于下限)的函数 FunctionBoundaryGMD(),由用户自己编写,其形式为:

```
template <class _Ty>
void FunctionBoundaryGMD(int j, valarray<_Ty>& x,
                                valarray<_Ty>& y)
{
    switch(j)
    {
      case 0: {
                  y[0] = c0;
                  y[1] = d0;
                  break;
              }
       case 1:{
                  y[0] = c1(x0)的表达式;
                  y[1] = d1(x0)的表达式;
                  break;
                }
            ⋮
      case n-1:{
                  y[0] = cn-1(x0,x1,⋯,xn-2)的表达式;
                  y[1] = dn-1(x0,x1,⋯,xn-2)的表达式;
                  break;
                }
        default: { }
    }
 }
```

(2) 本函数还要调用计算被积函数 $f(x_0,x_1,\cdots,x_{n-1})$值的函数 FunctionValueGMD(),由用户自己编写,其形式为:

```
template <class _Ty>
_Ty FunctionValueGMD(valarray<_Ty>& x)
{
    double z;
    ……
    z = 被积函数 f(x0,x1,⋯,xn-1)的表达式;
    return(z);
```

}

(3) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用高斯求积法计算三重积分

$$S=\int_0^1 dx\int_0^{\sqrt{1-x^2}} dy\int_{\sqrt{x^2+y^2}}^{\sqrt{2-x^2-y^2}} z^2 dz$$

其中 $n=3$。若将变量 x,y 和 z 分别用 x_0,x_1 和 x_2 表示，则有

$c_0=0.0$，　　$d_0=1.0$

$c_1(x_0)=0.0$，　　$d_1(x_0)=\sqrt{1-x_0^2}$

$c_2(x_0,x_1)=\sqrt{x_0^2+x_1^2}$，　　$d_2(x_0,x_1)=\sqrt{2-x_0^2-x_1^2}$

及

$$f(x_0,x_1,x_2)=x_2^2$$

设将每一层的积分区间均分为 4 个子区间，即 $js_0=js_1=js_2=4$。

主函数程序、计算各层积分上下限的函数程序及计算被积函数值的函数示例程序如下：

```
//IntegralGaussMD.cpp  多重高斯法求积

#include <iostream>          //输入输出流头文件
#include "Integral.h"        //积分头文件
using namespace std;         //名字空间

void main(void)
{
  long double js[3] = {4,4,4};
  valarray<long double> vjs(js, 3);
  long double dValue = IntegralGaussMD(vjs);
  cout << "IntegralGaussMD()" << endl << endl;
  cout << "dValue = " << dValue << endl << endl;
}

//计算各层积分上、下限值
template <class _Ty>
void FunctionBoundaryGMD(int j, valarray<_Ty>& x, valarray<_Ty>& y)
{
  switch(j)
{
    case 0:    {
                 y[0] = 0.0;
                 y[1] = 1.0;
                 break;
           }
     case 1:    {
                 y[0] = 0.0;
                 y[1] = sqrt(1.0 - x[0] * x[0]);
```

```
                    break;
                }
            case 2:    {
                    _Ty q = x[0] * x[0] + x[1] * x[1];
                    y[0] = sqrt(q);
                    y[1] = sqrt(2.0 - q);
                    break;
                    }
    }
}

//计算被积函数值
template <class _Ty>
_Ty FunctionValueGMD(valarray<_Ty>& x)
{
_Ty dValue;
dValue = x[2] * x[2];
   return(dValue);
}
```

8. 示例结果

```
IntegralGaussMD()

dValue = 0.382944
```

10.2.14 二重连分式法求积

1. 功能

用连分式法计算二重积分

$$S=\int_a^b \mathrm{d}x\int_{y_0(x)}^{y_1(x)} f(x,y)\mathrm{d}y$$

2. 算法

首先将二重积分化为两个单积分

$$S(x)=\int_{y_0(x)}^{y_1(x)} f(x,y)\mathrm{d}y \text{ 与 } S=\int_a^b S(x)\mathrm{d}x$$

然后利用连分式法计算每个单积分。关于用连分式法计算单积分的详细叙述见本章10.2.2节的算法说明。

计算二重积分的步骤如下。

(1) 固定一个 x，设为 $\overline{x}$，用连分式法计算单积分

$$S(\overline{x})=\int_{y_0(\overline{x})}^{y_1(\overline{x})} f(\overline{x},y)\mathrm{d}y$$

(2) 利用(1)中所计算得到的一系列 $S(\overline{x})$ 值，再利用连分式法计算 $S=\int_a^b S(x)\mathrm{d}x$。

3. 函数原型

```
//二重连分式法求积
```

```
template <class _Ty>
_Ty IntegralFraction2D(_Ty a, _Ty b, _Ty eps);
```

4. 参数

a:_Ty 类型变量。积分变量 x 的积分下限。

b:_Ty 类型变量。积分变量 x 的积分上限。要求 $b>a$。

eps:_Ty 类型变量。积分精度要求。

5. 返回值

_Ty 类型的函数值 $f(t)$。

6. 函数说明

(1) 本函数需要调用计算上、下限 $y_1(x)$与 $y_0(x)$(要求 $y_1(x)>y_0(x)$)的函数 FunctionBoundaryF2D(),由用户自己编写,其形式为:

```
template <class _Ty>
void FunctionBoundaryF2D(_Ty x, valarray<_Ty>& y)
{
  y[0] = 下限 y0(x)的表达式;
  y[1] = 上限 y1(x)的表达式;
}
```

(2) 本函数还要调用计算被积函数 $f(x,y)$值的函数 FunctionValueF2D(),由用户自己编写,其形式为:

```
template <class _Ty>
_Ty FunctionValueF2D(_Ty x, _Ty y)
{
  double z;
  z = 被积函数 f(x,y)的表达式;
  return(z);
}
```

(3) 所有输入和输出的数据都应使用浮点类型。

7. 示例

用连分式法计算二重积分

$$S=\int_0^1 dx\int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} e^{x^2+y^2}\,dy$$

其中 $a=0.0, b=1, y_0(x)=-\sqrt{1-x^2}, y_1(x)=\sqrt{1-x^2}, f(x,y)=e^{x^2+y^2}$,取 $eps=0.00005$。

主函数程序、计算上下限 $y_1(x)$与 $y_0(x)$的函数程序以及计算被积函数 $f(x,y)$的函数示例程序如下:

```
//IntegralFraction2D.cpp  二重连分式法求积

#include <iostream>          //输入输出流头文件
#include "Integral.h"        //积分头文件
using namespace std;         //名字空间

void main(void)
{
```

```
  long double eps(0.00005), a(0), b(1);
  long double dValue = IntegralFraction2D(a, b, eps);
  cout << "IntegralSimpson1D()" << endl << endl;
  cout << "dValue = " << dValue << endl << endl;
}

//计算上、下限 y1(x)与 y0(x)
template <class _Ty>
void FunctionBoundaryF2D(_Ty x, valarray<_Ty>& y)
{
  y[1] = sqrt(1.0 - x * x);
  y[0] = - y[1];
}

//计算被积函数 f(x,y)值
template <class _Ty>
_Ty FunctionValueF2D(_Ty x, _Ty y)
{
_Ty dValue;
dValue = exp(x * x + y * y);
   return(dValue);
}
```

8. 示例结果

```
IntegralSimpson1D()

dValue = 2.69911
```

10.2.15 多重蒙特卡洛法求积

1. 功能

用蒙特卡洛(Monte Carlo)法计算多重积分

$$S=\int_a^b\int_{a_1}^{b_1}\cdots\int_{a_{n-1}}^{b_{n-1}} f(x_0x_1,\cdots,x_{n-1})\mathrm{d}x_0\mathrm{d}x_1\cdots\mathrm{d}x_{n-1}$$

2. 算法

取 0 到 1 之间均匀分布的随机数点列

$$(t_0^{(i)},t_1^{(i)},\cdots,t_{n-1}^{(i)}),\quad i=0,1,\cdots,m-1$$

并令

$$x_j^{(i)}=a_j+(b_j-a_j)t_j^{(i)},\quad j=0,1,\cdots,n-1$$

只要 m 足够大,则有

$$S=\frac{1}{m}\Big[\sum_{i=0}^{m-1}f(x_0^{(i)},x_1^{(i)},\cdots,x_{n-1}^{(i)})\Big]\prod_{j=0}^{n-1}(b_j-a_j)$$

在本函数中,取 m=10000。

3. 函数原型

```
//多重蒙特卡洛法求积
template <class _Ty >
_Ty IntegralMonteCarlo2D(valarray<_Ty>& a, valarray<_Ty>& b);
```

4. 参数

a：_Ty 类型一维数组，长度为 n。存放积分的下限值 $a_0, a_1, \cdots, a_{n-1}$。

b：_Ty 类型一维数组，长度为 n。存放积分的上限值 $b_0, b_1, \cdots, b_{n-1}$。

5. 返回值

_Ty 类型积分值。

6. 函数说明

(1) 本函数需要调用产生 0 到 1 之间均匀分布随机数的 rand_01_One()函数，请参阅第 3 章 3.2.2 节。

(2) 本函数要调用计算被积函数 $f(x_0, x_1, \cdots, x_{n-1})$值的函数 FunctionValueMC2D()，由用户自己编写，其形式为：

```
template <class _Ty>
_Ty FunctionValueMC2D(int n, valarray<_Ty>& x)
{
  double y;
  y = 被积函数 f(x0, x1, ...xn-1)的表达式;
  return(y);
}
```

(3) 所有输入和输出的数据都应使用浮点类型，最好为 double 型。

7. 示例

用蒙特卡洛法计算三重积分

$$x = \int_1^2 \int_1^2 \int_1^2 (x_0^2 + x_1^2 + x_2^2)\,dx_0\,dx_1\,dx_2$$

主函数程序及计算被积函数值的函数示例程序如下：

```
//IntegralMonteCarlo2D.cpp  多重蒙特卡洛法求积

#include <iostream>          //输入输出流头文件
#include "Integral.h"        //积分头文件
using namespace std;         //名字空间

void main(void)
{
double a[3] = {1, 1, 1};
   double b[3] = {2, 2, 2};

valarray<double> da(a,3);
valarray<double> db(b,3);

double dValue = IntegralMonteCarlo2D(da, db);    //调用求积函数
```

```
cout << "IntegralMonteCarlo2D()" << endl << endl;
cout << "dValue = " << dValue << endl << endl;
}

//计算被积函数值
template <class _Ty>
_Ty FunctionValueMC2D(int n, valarray<_Ty>& x)
{
  _Ty dValue(0);
  for(int i = 0; i<n; i + + ) dValue = dValue + x[i] * x[i];
   return(dValue);
}
```

8. 示例结果

```
IntegralMonteCarlo2D()

dValue = 6.97043
```

第 11 章　常微分方程(组)求解

本章介绍有关的各种常微分方程(组)求解算法，每种算法都用 C++语言以函数的形式实现。11.1 节介绍常微分方程的基本概念，以及求解常微分方程相关算法程序原型的头文件，11.2 节给出了计算常微分方程(组)的使用示例及相关说明。

11.1　常微分方程与头文件

本节介绍常微分方程的基本概念，以及常微分方程算法和函数程序原型，其算法和函数程序原型包含在头文件 ordinaryDifferentialEguation. h 中，函数的定义内容在 ordinaryDifferentialEguation. inl 文件中。

本章介绍的常微分方程(组)算法及程序包括：

- 定步长全区间积分的欧拉法；
- 变步长积分欧拉法；
- 全区间定步长积分维梯法；
- 全区间定步长积分龙格一库塔法；
- 变步长积分龙格一库塔法；
- 变步长积分一步基尔法；
- 全区间变步长积分基尔法；
- 变步长积分默森法；
- 积分一步连分式法；
- 全区间积分连分式法；
- 全区间积分双边法；
- 全区间积分阿当姆斯预报一校正法；
- 全区间积分哈明法；
- 积分一步特雷纳法；
- 全区间积分特雷纳法；
- 积分刚性方程组吉尔法；
- 二阶微分方程边值问题数值解法。

11.1.1　基本概念

1. *微分方程*

对于某类数学问题，其中某个变量和其他变量之间的函数依赖关系是未知的，但是这个未知的函数关系以及它的某些阶的导数(或微分)连同自变量都由一个已知的方程联系在一起。这样的方程称为“微分方程”。

2. 常微分方程

如果微分方程中未知函数是一元的，那么对应的微分方程称为“常微分方程”。

常微分方程是联系自变量 x，未知函数 y 和它的某些阶导数 $\frac{dy}{dx},\frac{d^2y}{dx^2},\cdots,\frac{d^ny}{dx^n}$ 的关系式：

$$F\left(x,y,\frac{dy}{dx},\frac{d^2y}{dx^2},\cdots,\frac{d^ny}{dx^n}\right)=0$$

3. 微分方程的阶数

方程中出现的最高阶导数的阶数称为这个微分方程的阶。例如，$y''+y'^2=x^4$ 是二阶常微分方程。

4. 微分方程的解

使常微分方程成为恒等式的变量之间的关系式都是该常微分方程的解。如果关系式是隐式的，这种解又称为积分。如果常微分方程的解是数值的，那么称该解为“常微分方程数值解”。本章介绍的算法都是求常微分方程的数值解。

5. 微分方程的通解

如果在微分方程的解式中，所含的独立的任意常数（如果一个解中的常数可取任意值，称它为任意常数）的个数等于这个微分方程的阶数，那么这解式称为微分方程的“通解”。n 阶微分方程的通解表达式中含有 n 个彼此独立的任意常数。

6. 微分方程的特解

相对于通解而言，微分方程的每一个解称为“特解”。

7. 初值问题

如果在自变量的某值给出适当个数的附加条件，用来确定微分方程的特解，那么这样的问题称为“初值问题”。

8. 边值问题

如果在自变量一个以上的值给出适当个数的附加条件，用来确定微分方程的特解，那么这样的问题称为“边值问题”。

11.1.2 OrdinaryDifferentialEguation.h

```
//OrdinaryDifferentialEguation.h    求解常微分方程(组)头文件
// Ver 1.0.0.0
// 版权所有(C) 2002
// 版权所有(C) 何渝, 2002
// 最后修改：2002.9.4

#ifndef _ORDINARYDIFFERENTIALEGUATION_H    //避免多次编译
#define _ORDINARYDIFFERENTIALEGUATION_H

#include <LinearEquation.h>                //线性方程组头文件
#include <valarray>                        //模板类数组的标准头文件
#include <Matrix.h>                        //模板类矩阵头文件
#include <comm.h>                          //公共头文件
using namespace std;                       //名字空间
```

```
//定步长全区间积分的欧拉法
template <class _Ty>
void ODE_EulerContentStep(_Ty t, valarray<_Ty>& y, _Ty h, int k, matrix<_Ty>& z);

//变步长积分欧拉法
template <class _Ty>
void ODE_EulerVariationalStep(_Ty t, _Ty h, valarray<_Ty>& y, _Ty eps);

//全区间定步长积分维梯法
template <class _Ty>
void ODE_WittyContentStep(_Ty t, valarray<_Ty>& y, _Ty h, int k, matrix<_Ty>& z);

//全区间定步长积分龙格-库塔法
template <class _Ty>
void ODE_RungeKuttaContentStep(_Ty t, valarray<_Ty>& y, _Ty h, int k, matrix<_Ty>& z);

//变步长积分龙格-库塔法
template <class _Ty>
void ODE_RungeKuttaVariationalStep(_Ty t, _Ty h, valarray<_Ty>& y, _Ty eps);

//变步长积分一步基尔法
template <class _Ty>
void ODE_GillVariationalStep(_Ty t, _Ty h, valarray<_Ty>& y,_Ty eps, valarray<_Ty>& q);

//全区间变步长积分基尔法
template <class _Ty>
void ODE_GillVariationalStepInterval(_Ty t, _Ty h, valarray<_Ty>& y,_Ty eps, int k, matrix<_Ty>& z);

//变步长积分默森法
template <class _Ty>
void ODE_MersonVariationalStep(_Ty t, _Ty h, int n, valarray<_Ty>& y,_Ty eps, int k, matrix<_Ty>& z);

//积分一步连分式法
template <class _Ty>
void ODE_Fraction(_Ty t, _Ty h, valarray<_Ty>& y, _Ty eps);

//全区间积分连分式法
template <class _Ty>
void ODE_FractionInterval(_Ty t, _Ty h, valarray<_Ty>& y,_Ty eps, int k, matrix<_Ty>& z);

//全区间积分双边法
template <class _Ty>
void ODE_BothSidesInterval(_Ty t, _Ty h, valarray<_Ty>& y,_Ty eps, int k, matrix<_Ty>& z);
```

```
//全区间积分阿当姆斯预报-校正法
template <class _Ty>
void ODE_AdamsInterval(_Ty t, _Ty h, valarray<_Ty>& y,_Ty eps, int k, matrix<_Ty>& z);

//全区间积分哈明法
template <class _Ty>
void ODE_HammingInterval(_Ty t, _Ty h, valarray<_Ty>& y,_Ty eps, int k, matrix<_Ty>& z);

//积分一步特雷纳法
template <class _Ty>
void ODE_Treanor(_Ty t, _Ty h, valarray<_Ty>& y);

//全区间积分特雷纳法
template <class _Ty>
void ODE_TreanorInterval(_Ty t, _Ty h, valarray<_Ty>& y,int k, matrix<_Ty>& z);

//积分刚性方程组吉尔法
template <class _Ty>
int ODE_Gear(_Ty a, _Ty b, _Ty hmin, _Ty hmax, _Ty h, _Ty eps,
          valarray<_Ty>& y0, int k, valarray<_Ty>& t, matrix<_Ty>& z);

//二阶微分方程边值问题数值解法
template <class _Ty>
void ODE_LinearBoundaryValude(_Ty a, _Ty b, _Ty ya, _Ty yb, int n, valarray<_Ty>& y);

#include "OrdinaryDifferentialEguation.inl"     //类及相关函数的定义头文件

#endif     // _ORDINARYDIFFERENTIALEGUATION_H
```

11.2 常微分方程(组)计算方法

11.2.1 定步长全区间积分的欧拉法

1. 功能

用改进的欧拉(Euler)公式对一阶微分方程组进行定步长全区间积分。

2. 算法

设一阶微分方程组

$$\begin{cases} y_0' = f_0(t, y_0, y_1, \cdots, y_{n-1}), y(t_0) = y_{00} \\ y_1' = f_1(t, y_0, y_1, \cdots, y_{n-1}), y_1(t_0) = y_{10} \\ \vdots \\ y_{n-1}' = f_{n-1}(t, y_0, y_1, \cdots, y_{n-1}), y_{n-1}(t_0) = y_{n-1,0} \end{cases}$$

已知 $t = t_{j-1}$ 点的函数值 $y_{i,j-1}(i = 0,1,\cdots,n-1)$，求 $t_j = t_{j-1} + h$ 点的函数值 $y_{ij}(i = 0,1,\cdots,$

$n-1$)。

改进的欧拉公式为：

$$p_i = y_{i,j-1} + hf_i(t_{j-1}, y_{0,j-1}, \cdots, y_{n-1,j-1}), \quad i = 0,1,\cdots,n-1$$

$$q_i = y_{i,j-1} + hf_i(t_{j-1}, p_0, p_1, \cdots, p_{n-1}), \quad i = 0,1,\cdots,n-1$$

$$y_{ij} = \frac{1}{2}(p_i + q_i), \quad i = 0,1,\cdots,n-1$$

3. 函数原型

```
//定步长全区间积分的欧拉法
template <class _Ty>
void ODE_EulerContentStep(_Ty t, valarray<_Ty>& y, _Ty h, int k, matrix<_Ty>& z);
```

4. 参数

t：_Ty 类型变量。对微分方程进行积分的起始点 t_0。

y：valarray<_Ty>类型引用，长度为 n，存放 n 个未知函数 y_i 在起点 t_0 处的函数值(即初值)$y_i(t_0)(i=0,1,\cdots,n-1)$。返回时，这些初值存放在二维数组 z 的第零列中，即 $z(i,0)=y_i(t_0)(i=0,1,\cdots,n-1)$。

h：_Ty 类型。积分步长。

k：整型变量。积分步长数(包括起始点这一步)。

z：matrix<_Ty>引用，体积为 $n\times k$。返回 k 个积分点(包括起始点)上的未知函数值，即

$$z(i,j) = y_{ij}, i = 0,1,\cdots,n-1; j = 0,1,\cdots,k-1$$

其中，$z(i,0)(i=0,1,\cdots,n-1)$为初值 $y_i(t_0)$。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t, y_0, y_1, \cdots, y_{n-1})(i=0,1,\cdots,n-1)$的函数 FunctionValueECS()，由用户自己编写，其形式如下：

```
template <class _Ty>
void FunctionValueECS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();     //微分方程组中方程的个数，也是未知函数的个数
      ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
      ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为：

$$\begin{cases} y_0' = y_1, y_0(0) = -1.0 \\ y_1' = -y_0, y_1(0) = 0.0 \\ y_2' = -y_2, y_2(0) = 1.0 \end{cases}$$

用改进欧拉公式计算当步长 $h=0.01$ 时，11 个积分点(包括起始点)

$$t_i = i * h, i = 0,1,\cdots,10$$

上的未知函数的近似值 $y_{0i}, y_{1i}, y_{2i}(i=0,1,\cdots,10)$。其中起始点 $t=0.0, n=3, k=11, h=0.01$。

主函数程序及计算微分方程组中各方程组右端函数值的函数示例程序如下：

```
//ODE_EulerContentStep.cpp                    定步长全区间积分的欧拉法

#include <iostream>                           //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"     //求解常微分方程(组)头文件
using namespace std;                          //名字空间

void main(void)
{
    double y[3] = {-1.0, 0.0, 1.0}, t(0.0), h(0.01);
    valarray<double> vy(y, 3);
    matrix<double> vz(3,11);

    ODE_EulerContentStep(t,vy,h,11,vz);

    cout.setf(ios::fixed);                    //输出数据为定点法
    cout.precision(6);                        //精度 6 位

    for(int i=0; i<11; i++)
    {
      double x = h * i;
      cout << "t = " << x << endl;
      for(int j=0; j<3; j++)
          cout << " y(" << j << ") = " << vz(j, i) << "\t";
      cout << endl;
    }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueECS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
        int n = y.size();      //微分方程组中方程的个数,也是未知函数的个数
        d[0] = y[1];
        d[1] = -y[0];
        d[2] = -y[2];
}
```

8. 示例结果

```
t = 0.000000
    y(0) = -1.000000     y(1) = 0.000000     y(2) = 1.000000
```

```
t = 0.010000
    y(0) = -0.999950    y(1) = 0.010000    y(2) = 0.990050
t = 0.020000
    y(0) = -0.999800    y(1) = 0.019999    y(2) = 0.980199
t = 0.030000
    y(0) = -0.999550    y(1) = 0.029996    y(2) = 0.970446
t = 0.040000
    y(0) = -0.999200    y(1) = 0.039990    y(2) = 0.960790
t = 0.050000
    y(0) = -0.998750    y(1) = 0.049980    y(2) = 0.951230
t = 0.060000
    y(0) = -0.998200    y(1) = 0.059965    y(2) = 0.941765
t = 0.070000
    y(0) = -0.997551    y(1) = 0.069944    y(2) = 0.932395
t = 0.080000
    y(0) = -0.996802    y(1) = 0.079916    y(2) = 0.923118
t = 0.090000
    y(0) = -0.995953    y(1) = 0.089880    y(2) = 0.913933
t = 0.100000
    y(0) = -0.995004    y(1) = 0.099835    y(2) = 0.904839
```

本问题的解析解为

$$\begin{cases} y_0 = -\cos t \\ y_1 = \sin t \\ y_2 = \mathrm{e}^{-t} \end{cases}$$

11.2.2 变步长积分欧拉法

1. 功能

用变步长欧拉(Euler)方法对一阶微分方程组积分一步。

2. 算法

同本章 11.2.1 节的算法。

根据改进的欧拉公式以 h 为步长，由 $y_{i,j-1}(i=0,1,\cdots,n-1)$ 计算 $y_{ij}^{(h)}$，再以 $h/2$ 为步长，由 $y_{i,j-1}$ 跨两步计算 $y_{ij}^{(h/2)}$。此时，若

$$\max_{0\leqslant i\leqslant n-1} \left| y_{ij}^{(h/2)} - y_{ij}^{(h)} \right| < \varepsilon$$

则停止计算，取 $y_{ij}^{(h/2)}$ 作为 $y_{ij}(i=0,1,\cdots,n-1)$；否则，将步长折半再进行计算。

上述过程一直作到满足条件

$$\max_{0\leqslant i\leqslant n-1} \left| y_{ij}^{(h/2^m)} - y_{ij}^{(h/2^{m-1})} \right| < \varepsilon$$

为止。最后可取 $y_{ij} = y_{ij}^{(h/2^m)}$，$i=0,1,\cdots,n-1$，其中 ε 为预先给定的精度要求。

3. 函数原型

```
//变步长积分欧拉法
template <class _Ty>
void ODE_EulerVariationalStep(_Ty t, _Ty h, valarray<_Ty>& y, _Ty eps);
```

4. 参数

t:_Ty 类型变量。积分一步的起始点。

h:_Ty 类型变量。积分步长。

y:valarray<_Ty>类型引用,长度为 n。存放起点 t 处的 n 个未知函数值 $y_i(t)(i=0,1,\cdots,n-1)$;返回时存放 $t+h$ 点处的 n 个未知函数值 $y_i(t+h)(i=0,1,\cdots,n-1)$。

eps:_Ty 类型变量。积分的精度要求。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,\cdots,n-1)$的函数 FunctionValueECS(),由用户自己编写,其形式如下:

```
template <class _Ty>
void FunctionValueECS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();     //微分方程组中方程的个数,也是未知函数的个数
        ……
    d[0]=f0(t,y0,y1,…,yn-1)的表达式
    d[1]=f1(t,y0,y1,…,yn-1)的表达式
        ……
    d[n-1]=fn-1(t,y0,y1,…,yn-1)的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y_0' = y_1, y_0(0) = -1.0 \\ y_1' = -y_0, y_1(0) = 0.0 \\ y_2' = -y_2, y_2(0) = 1.0 \end{cases}$$

用积分一步的变步长欧拉方法计算当步长 $h=0.01$ 时各积分点

$$t_i = i \times h, i = 0,1,\cdots,10$$

上的未知函数的近似值 $y_{0i}, y_{1i}, y_{2i}(i=0,1,\cdots,10)$。取 $\varepsilon=0.00001$。

主函数程序及计算微分方程组右端函数值的函数示例程序如下:

```
//ODE_EulerVariationalStep.cpp                变步长积分欧拉法

#include <iostream>                           //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"     //求解常微分方程(组)头文件
using namespace std;                          //名字空间

void main(void)
{
    double y[3] = {-1.0, 0.0, 1.0}, t(0.0), h(0.01), eps(0.00001);
    valarray<double> vy(y, 3);
```

```
    cout.setf(ios::fixed);                    //输出数据为定点法
    cout.precision(6);                        //精度 6 位

    cout << "t = " << t << endl;
    for(int i = 0; i<3; i++)
      cout << " y(" << i << ") = " << y[i] << "\t";
    cout << endl;

    for(int j = 1; j<11; j++)
    {
      ODE_EulerVariationalStep(t,h,vy,eps);
      t = t + h;
      cout << "t = " << t << endl;
      for(int i = 0; i<3; i++)
          cout << " y(" << i << ") = " << vy[i] << "\t";
      cout << endl;
    }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueEVS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
int n = y.size();          //微分方程组中方程的个数,也是未知函数的个数
      d[0] = y[1];
      d[1] = -y[0];
      d[2] = -y[2];
}
```

8. 示例结果

```
t = 0.000000
    y(0) = -1.000000    y(1) = 0.000000    y(2) = 1.000000
t = 0.010000
    y(0) = -0.999950    y(1) = 0.010000    y(2) = 0.990050
t = 0.020000
    y(0) = -0.999800    y(1) = 0.019999    y(2) = 0.980199
t = 0.030000
    y(0) = -0.999550    y(1) = 0.029996    y(2) = 0.970446
t = 0.040000
    y(0) = -0.999200    y(1) = 0.039990    y(2) = 0.960790
t = 0.050000
    y(0) = -0.998750    y(1) = 0.049979    y(2) = 0.951230
t = 0.060000
    y(0) = -0.998201    y(1) = 0.059964    y(2) = 0.941765
t = 0.070000
```

```
    y(0) = -0.997551    y(1) = 0.069943    y(2) = 0.932394
t = 0.080000
    y(0) = -0.996802    y(1) = 0.079915    y(2) = 0.923117
t = 0.090000
    y(0) = -0.995953    y(1) = 0.089879    y(2) = 0.913932
t = 0.100000
    y(0) = -0.995004    y(1) = 0.099834    y(2) = 0.904838
```

本问题的解析解为

$$\begin{cases} y_0 = -\cos t \\ y_1 = \sin t \\ y_2 = \mathrm{e}^{-t} \end{cases}$$

11.2.3 全区间定步长积分维梯法

1. 功能

用定步长维梯(witty)方法对一阶微分方程进行全区间积分。

2. 算法

设一阶微分方程组为

$$\begin{cases} y_0 = f_0(t, y_0, y_1, \cdots, y_{n-1}), & y_0(t_0) = y_{00} \\ y_1 = f_1(t, y_0, y_1, \cdots, y_{n-1}), & y_1(t_0) = y_{10} \\ \quad\vdots & \quad\vdots \\ y_{n-1} = f_{n-1}(t, y_0, y_1, \cdots, y_{n-1}), & y_{n-1}(t_0) = y_{n-1,0} \end{cases}$$

用维梯方法由 t_j 积分一步到 $t_{j+1}=t_j+h$ 的计算公式如下：

$$d_{i0} = f_i(t_0, y_{00}, y_{10}, \cdots, y_{n-1,0}), i = 0,1,\cdots,n-1$$

$$y_{i,j+\frac{1}{2}} = y_{ij} + \frac{1}{2}hd_{ij}, i = 0,1,\cdots,n-1$$

$$q_i = f_i\left(t_j + \frac{1}{2}, y_{0,j+\frac{1}{2}}, y_{1,j+\frac{1}{2}}, \cdots, y_{n-1,j+\frac{1}{2}}\right), i = 0,1,\cdots,n-1$$

$$y_{i,j+1} = y_{ij} + hq_i, i = 0,1,\cdots,n-1$$

$$d_{i,j+1} = 2q_i - d_{ij}, i = 0,1,\cdots,n-1$$

3. 函数原型

```
//全区间定步长积分维梯法
template <class _Ty>
void ODE_WittyContentStep(_Ty t, valarray<_Ty>& y, _Ty h, int k, matrix<_Ty>& z);
```

4. 参数

t：_Ty 类型变量。积分起始点 t_0。

y：valarray<_Ty>类型引用，长度为 n。起始点 t 处的 n 个未知函数值 $y_i(t_0)(i=0,1,\cdots,n-1)$(即初值)。返回时其值存放二维数组 z 的第零列中，即 $z(i,0)=y_i(t_0), i=0,1,\cdots,n-1$。

h：_Ty 类型变量。积分步长。

k：整型变量。积分步长数(包括起始点这一步)。

z：matrix<_Ty>引用，体积为 $n\times k$。返回 k 个积分点(包括起始点)上的未知函数值，即

$$z(i,j) = y_{ij}, i = 0,1,\cdots,n-1; j = 0,1,\cdots,k-1$$

其中 $z(i,0)=y_i(t_0)$ $(i=0,1,\cdots,n-1)$。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,\cdots,n-1)$的函数 FunctionValueWCS(),由用户自己编写,其形式如下:

```
template <class _Ty>
void FunctionValueWCS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();          //微分方程组中方程的个数,也是未知函数的个数
        ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
        ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y_0' = y_1, y_0(0) = -1.0 \\ y_1' = -y_0, y_1(0) = 0.0 \\ y_2' = -y_2, y_2(0) = 1.0 \end{cases}$$

用维梯方法计算当步长 $h=0.1$ 时,11 个积分点

$$t_i = i \times h, i = 0,1,\cdots,10$$

上的未知函数的近似值 $y_{0i},y_{1i},y_{2i}(i=0,1,\cdots,10)$。其中起始点 $t=0.0,n=3,k=11,h=0.1$。

主函数程序及计算微分方程右端函数值的函数示例程序如下:

```
//ODE_WittyContentStep.cpp                    全区间定步长积分维梯法

#include <iostream>                           //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"     //求解常微分方程(组)头文件
using namespace std;                          //名字空间

void main(void)
{
    double y[3] = {-1.0, 0.0, 1.0}, t(0.0), h(0.1);
    valarray<double> vy(y, 3);
    matrix<double> vz(3,11);

    ODE_WittyContentStep(t,vy,h,11,vz);

    cout.setf(ios::fixed);                    //输出数据为定点法
    cout.precision(6);                        //精度 6 位
```

```
    for(int i = 0; i<11; i ++ )
    {
      double x = h * i;
      cout << "t = " << x << endl;
      for(int j = 0; j<3; j ++ )
          cout << " y(" << j << ") = " << vz(j, i) << "\t";
      cout << endl;
    }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueWCS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();      //微分方程组中方程的个数,也是未知函数的个数
    d[0] = y[1];
    d[1] = - y[0];
    d[2] = - y[2];
}
```

8. 示例结果

```
t = 0.000000
    y(0) = -1.000000    y(1) = 0.000000    y(2) = 1.000000
t = 0.100000
    y(0) = -0.995000    y(1) = 0.100000    y(2) = 0.905000
t = 0.200000
    y(0) = -0.980000    y(1) = 0.199000    y(2) = 0.819000
t = 0.300000
    y(0) = -0.955200    y(1) = 0.296000    y(2) = 0.741200
t = 0.400000
    y(0) = -0.920800    y(1) = 0.390040    y(2) = 0.670760
t = 0.500000
    y(0) = -0.877192    y(1) = 0.480160    y(2) = 0.607048
t = 0.600000
    y(0) = -0.824768    y(1) = 0.565478    y(2) = 0.549350
t = 0.700000
    y(0) = -0.764096    y(1) = 0.645114    y(2) = 0.497178
t = 0.800000
    y(0) = -0.695745    y(1) = 0.718298    y(2) = 0.449915
t = 0.900000
    y(0) = -0.620437    y(1) = 0.784263    y(2) = 0.407195
t = 1.000000
    y(0) = -0.538893    y(1) = 0.842385    y(2) = 0.368476
```

本问题的解析解为

$$\begin{cases} y_0 = -\cos t \\ y_1 = \sin t \\ y_2 = \mathrm{e}^{-t} \end{cases}$$

11.2.4 全区间定步长积分龙格一库塔法

1. 功能

用定步长四阶龙格一库塔(Runge-Kutta)法对一阶微分方程进行全区间积分。

2. 算法

设一阶微分方程组为

$$\begin{cases} y_0' = f_0(t, y_0, y_1, \cdots, y_{n-1}), & y_0(t_0) = y_{00} \\ y_1' = f_1(t, y_0, y_1, \cdots, y_{n-1}), & y_1(t_0) = y_{10} \\ \quad\vdots & \quad\vdots \\ y_{n-1}' = f_{n-1}(t, y_0, y_1, \cdots, y_{n-1}), & y_{n-1}(t_0) = y_{n-1,0} \end{cases}$$

由 t_j 积分一步到 $t_{j+1}=t_j+h$ 的四阶龙格一库塔方法的计算公式为

$$k_{0i} = f_i(t_j, y_{0j}, y_{j1}, \cdots, y_{n-1,j}), i = 0,1,\cdots,n-1$$

$$k_{1i} = f_i\left(t_j + \frac{h}{2}, + y_{0j} + \frac{h}{2}k_{00}, \cdots, y_{n-1,j} + \frac{h}{2}k_{0,n-1}\right), i = 0,1,\cdots,n-1$$

$$k_{2i} = f_i\left(t_j + \frac{h}{2}, + y_{0j} + \frac{h}{2}k_{10}, \cdots, y_{n-1,j} + \frac{h}{2}k_{1,n-1}\right), i = 0,1,\cdots,n-1$$

$$k_{3i} = f_i(t_j + h, y_{0j} + hk_{20}, \cdots, y_{n-1,j} + hk_{2,n-1}), i = 0,1,\cdots,n-1$$

$$y_{i,j+1} = y_{ij} + \frac{h}{6}(k_{0i} + 2k_{1i} + 2k_{2i} + k_{3i}), i = 0,1,\cdots,n-1$$

3. 函数原型

```
//全区间定步长积分龙格一库塔法
template <class _Ty>
void ODE_RungeKuttaContentStep(_Ty t, valarray<_Ty>& y, _Ty h, int k, matrix<_Ty>& z);
```

4. 参数

t:_Ty 类型变量。积分起始点 t_0。

y:valarray<_Ty>类型引用,长度为 n。存放 n 个未知函数在起始点 t 处的函数值(即初值)$y_i(t)(i=0,1,\cdots,n-1)$。返回时,其值存放二维数组 z 的第零列中,即

$$z(i,0) = y_i(t), i = 0,1,\cdots,n-1$$

h:_Ty 类型变量。积分步长。

k:整型变量。积分步长数(包括起始点这一步)。

z:matrix<_Ty>引用,体积为 $n\times k$。返回 k 个积分点上的 n 个未知函数值,即

$$z(i,j) = y_i(t_j), i = 0,1,\cdots,n-1; j = 0,1,\cdots,k-1$$

其中,$z(i,0)(i=0,1,\cdots,n-1)$为给定的初值。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t, y_0, y_1, \cdots, y_{n-1})(i=0,1,\cdots,n-1)$的函数 FunctionValueRKCS(),由用户自己编写,其形式如下:

```
template <class _Ty>
void FunctionValueRKCS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();          //微分方程组中方程的个数,也是未知函数的个数
        ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
        ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y_0' = y_1, y_0(0) = -1.0 \\ y_1' = -y_0, y_1(0) = 0.0 \\ y_2' = -y_2, y_2(0) = 1.0 \end{cases}$$

用四阶龙格一库塔法计算当 $h=0.01$ 时,11 个积分点

$$t_i = i \times h, i = 0,1,\cdots,10$$

上的未知函数的近似值 $y_{0i}, y_{1i}, y_{2i}(i=0,1,\cdots,10)$。其中起始点 $t=0.0, h=0.01, n=3, k=11$。

主函数程序及计算微分方程组中各方程右端函数值的函数示例程序如下:

```
//ODE_RungeKuttaContentStep.cpp                全区间定步长积分龙格一库塔法

#include <iostream>                              //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"        //求解常微分方程(组)头文件
using namespace std;                             //名字空间

void main(void)
{
    double y[3] = {-1.0, 0.0, 1.0}, t(0.0), h(0.01);
    valarray<double> vy(y, 3);
    matrix<double> vz(3,11);

    ODE_RungeKuttaContentStep(t,vy,h,11,vz);

    cout.setf(ios::fixed);                       //输出数据为定点法
    cout.precision(6);                           //精度 6 位

    for(int i=0; i<11; i++)
    {
      double x = h * i;
      cout << "t = " << x << endl;
      for(int j=0; j<3; j++)
          cout << "y(" << j << ") = " << vz(j, i) << "\t";
```

```
        cout << endl;
    }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueRKCS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size(); //微分方程组中方程的个数,也是未知函数的个数

    d[0] = y[1];
    d[1] = -y[0];
    d[2] = -y[2];
}
```

8. 示例结果

```
t = 0.000000
    y(0) = -1.000000    y(1) = 0.000000    y(2) = 1.000000
t = 0.010000
    y(0) = -0.999950    y(1) = 0.010000    y(2) = 0.990050
t = 0.020000
    y(0) = -0.999800    y(1) = 0.019999    y(2) = 0.980199
t = 0.030000
    y(0) = -0.999550    y(1) = 0.029996    y(2) = 0.970446
t = 0.040000
    y(0) = -0.999200    y(1) = 0.039989    y(2) = 0.960789
t = 0.050000
    y(0) = -0.998750    y(1) = 0.049979    y(2) = 0.951229
t = 0.060000
    y(0) = -0.998201    y(1) = 0.059964    y(2) = 0.941765
t = 0.070000
    y(0) = -0.997551    y(1) = 0.069943    y(2) = 0.932394
t = 0.080000
    y(0) = -0.996802    y(1) = 0.079915    y(2) = 0.923116
t = 0.090000
    y(0) = -0.995953    y(1) = 0.089879    y(2) = 0.913931
t = 0.100000
    y(0) = -0.995004    y(1) = 0.099833    y(2) = 0.904837
```

本问题的解析解为

$$\begin{cases} y_0 = -\cos t \\ y_1 = \sin t \\ y_2 = \mathrm{e}^{-t} \end{cases}$$

11.2.5 变步长积分龙格－库塔法

1. 功能

用变步长四阶龙格－库塔(Runge-Kutta)法对一阶微分方程组积分一步。

2. 算法

算法同本章 11.2.4 节。

根据四阶龙格－库塔公式，以 h 为步长，由 $y_{i,j-1}$ 计算 $y_{ij}^{(h)}(i=0,1,\cdots,n-1)$；再以 $h/2$ 为步长，由 $y_{i,j-1}$ 跨两步计算 $y_{ij}^{(h/2)}(i=0,1,\cdots,n-1)$。若满足条件

$$\max_{0\leqslant i\leqslant n-1}\left|y_{ij}^{(h/2)}-y_{ij}^{(h)}\right|<\varepsilon$$

则停止计算，取 $y_i(t_i)=y_{ij}^{(h/2)}(i=1,0,\cdots,n-1)$；否则，将步长折半再进行计算。

以上过程一直作到满足条件

$$\max_{0\leqslant i\leqslant n-1}\left|y_{ij}^{(h/2^m)}-y_{ij}^{(h/2^{m-1})}\right|<\varepsilon$$

为止。最后可得

$$y_i(t_j)=y_{ij}^{(h/2^m)},i=0,1,\cdots,n-1$$

其中 ε 为预先给定的精度要求。

3. 函数原型

```
//变步长积分龙格一库塔法
template <class _Ty>
void ODE_RungeKuttaVariationalStep(_Ty t, _Ty h,
                                   valarray<_Ty>& y, _Ty eps);
```

4. 参数

t：_Ty 类型变量。积分起始点。

h：_Ty 类型变量。积分步长。

y：valarray<_Ty>类型引用，长度为 n。存放起点 t 处的 n 个未知函数值 $y_i(t)(i=0,1,\cdots,n-1)$；返回时存放积分一步后 $t+h$ 点的 n 个未知函数值 $y_i(t+h)(i=0,1,\cdots,n-1)$。

eps：_Ty 类型变量。积分的精度要求。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,\cdots,n-1)$ 的函数 FunctionValueRKCS()，由用户自己编写，其形式如下：

```
template <class _Ty>
void FunctionValueRKCS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();          //微分方程组中方程的个数，也是未知函数的个数
        ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
        ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y_0' = y_1, y_0(0) = 0.0 \\ y_1' = -y_0, y_1(0) = 1.0 \end{cases}$$

用变步长四阶龙格－库塔法计算当 $h=0.1$ 时各积分点

$$t_i = i \times h, i = 0,1,\cdots,10$$

上的未知函数的近似值 $y_{0i}, y_{1i}(i=0,1,\cdots,10)$。取 $eps=0.00001$。

主函数程序及计算微分方程组右端函数值的函数示例程序如下：

```
//ODE_RungeKuttaVariationalStep.cpp                变步长积分龙格—库塔法

#include <iostream>                                //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"          //求解常微分方程(组)头文件
using namespace std;                               //名字空间

void main(void)
{
    double dy[2] = {0.0, 1.0}, t(0.0), h(0.1), eps(FLOATERROR);
    valarray<double> y(dy, 2);

    cout.setf(ios::fixed);                         //输出数据为定点法
    cout.precision(6);                             //精度 6 位

    cout << " t = " << t << "\t y[0] = " << y[0] << "\t y[1] = " << y[1] << endl;

    for(int i=1; i<11; i++)
    {
      ODE_RungeKuttaVariationalStep(t,h,y,eps);
      t = t + h;
      cout << " t = " << t << "\t y[0] = " << y[0] << "\t y[1] = " << y[1] << endl;
    }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueRKVS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();   //微分方程组中方程的个数,也是未知函数的个数

      d[0] = y[1];
      d[1] = -y[0];
}
```

8. 示例结果

```
t = 0.000000    y[0] = 0.000000    y[1] = 1.000000
t = 0.100000    y[0] = 0.099833    y[1] = 0.995004
t = 0.200000    y[0] = 0.198669    y[1] = 0.980067
t = 0.300000    y[0] = 0.295520    y[1] = 0.955336
t = 0.400000    y[0] = 0.389418    y[1] = 0.921061
t = 0.500000    y[0] = 0.479426    y[1] = 0.877583
t = 0.600000    y[0] = 0.564642    y[1] = 0.825336
t = 0.700000    y[0] = 0.644218    y[1] = 0.764842
t = 0.800000    y[0] = 0.717356    y[1] = 0.696707
t = 0.900000    y[0] = 0.783327    y[1] = 0.621610
t = 1.000000    y[0] = 0.841471    y[1] = 0.540302
```

本问题的解析解为

$$y_0 = \sin t$$
$$y_1 = \cos t$$

11.2.6 变步长积分一步基尔法

1. 功能

用变步长基尔(Gill)公式对一阶微分方程组积分一步。

2. 算法

设一阶微分方程组为

$$\begin{cases} y_0 = f_0(t, y_0, y_1, \cdots, y_{n-1}), y_0(t_0) = y_{00} \\ y_1 = f_1(t, y_0, y_1, \cdots, y_{n-1}), y_1(t_0) = y_{10} \\ \quad \vdots \\ y_{n-1} = f_{n-1}(t, y_0, y_1, \cdots, y_{n-1}), y_{n-1}(t_0) = y_{n-1,0} \end{cases}$$

由 t 点到 $t+h$ 点的基尔公式如下：

$$\begin{cases} k_{0i} = hf_i(t, y_0^{(0)}, y_1^{(0)}, \cdots, y_{n-1}^{(0)}) \\ k_{1i} = hf_i\left(t+\frac{h}{2}, y_0^{(1)}, y_1^{(1)}, \cdots, y_{n-1}^{(1)}\right) \\ k_{2i} = hf_i\left(t+\frac{h}{2}, y_0^{(2)}, y_1^{(2)}, \cdots, y_{n-1}^{(2)}\right) \\ k_{3i} = hf_i(t+h, y_0^{(3)}, y_1^{(3)}, \cdots, y_{n-1}^{(3)}) \end{cases} \quad i = 0, 1, \cdots, n-1$$

其中

$$\begin{cases} y_i^{(1)} = y_i^{(0)} + \frac{1}{2}(k_{0i} - 2q_i^{(0)}) \\ y_i^{(2)} = y_i^{(1)} + \left(1 - \sqrt{\frac{1}{2}}\right)(k_{1i} - q_i^{(1)}) \\ y_i^{(3)} = y_i^{(2)} + \left(1 + \sqrt{\frac{1}{2}}\right)(k_{2i} - q_i^{(2)}) \\ y_i^{(4)} = y_i^{(3)} + \frac{1}{6}(k_{3i} - 2q_i^{(3)}) \end{cases} \quad i = 0, 1, \cdots, n-1$$

式中

$$\begin{cases} q_i^{(1)} = q_i^{(0)} + 3\left[\frac{1}{2}(k_{0i} - 2q_i^{(0)})\right] - \frac{1}{2}k_{0i} \\ q_i^{(2)} = q_i^{(1)} + 3\left[\left(1-\sqrt{\frac{1}{2}}\right)(k_{1i} - q_i^{(1)})\right] - \left(1-\sqrt{\frac{1}{2}}\right)k_{1i} \\ q_i^{(3)} = q_i^{(2)} + 3\left[\left(1+\sqrt{\frac{1}{2}}\right)(k_{2i} - q_i^{(2)})\right] - \left(1+\sqrt{\frac{1}{2}}\right)k_{2i} \\ q_i^{(4)} = q_i^{(3)} + 3\left[\frac{1}{6}(k_{3i} - 2q_i^{(3)})\right] - \frac{1}{2}k_{3i} \end{cases}$$

$$i = 0,1,\cdots,n-1$$

其中：

$y_i^{(0)}$ 为 t 点的未知函数值 $y_i(t)$；

$y_i^{(4)}$ 为积分一步后，$t+h$ 点的未知函数值 $y_i(t+h)$；

$q_i^{(0)}$ 在起始值为 0，以后每积分一步，将 $q_i^{(4)}$ 作为下一步的 $q_i^{(0)}$。

这种方法具有抵消每一步中所积累的舍入误差的作用，可以提高精度。

本函数在积分一步的过程中，还具有自动选择步长的作用，即根据给定的精度要求，自动将步长等分，跨多步积分到 $t+h$。有关自动选择步长的方法参看本章 11.2.5 节的算法。

3. 函数原型

```
//变步长积分一步基尔法
template <class _Ty>
void ODE_GillVariationalStep(_Ty t, _Ty h, valarray<_Ty>& y,_Ty eps, valarray<_Ty>& q);
```

4. 参数

t:_Ty 类型变量。积分的起始点。

h:_Ty 类型变量。积分步长。

y:valarray<_Ty>类型引用，长度为 n，存放起始点 t 处的 n 个未知函数值 $y_i(t)(i=0,1,\cdots,n-1)$；返回时存放 $t+h$ 点处的 n 个未知函数值 $y_i(t+h)(i=0,1,\cdots,n-1)$。

eps:_Ty 类型变量。积分一步的精度要求。

k:整型变量。积分步长数(包括起始点这一步)。

q:valarray<_Ty>类型引用，长度为 n。在主函数第一次调用时，应赋值以 0，即 $q(i)=0(i=0,1,\cdots,n-1)$ 以后每调用一次本函数(即每积分一步)，将由本函数的返回值以便循环使用。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,\cdots,n-1)$ 的函数 FunctionValueRKCS()，由用户自己编写，其形式如下：

```
template <class _Ty>
void FunctionValueRKCS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();     //微分方程组中方程的个数，也是未知函数的个数
        ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
```

$d[1] = f_1(t, y_0, y_1, \cdots, y_{n-1})$ 的表达式

……

$d[n-1] = f_{n-1}(t, y_0, y_1, \cdots, y_{n-1})$ 的表达式

}

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y_0 = y_1, & y_0(0) = 0.0 \\ y_1 = -y_0, & y_1(0) = 1.0 \\ y_2 = -y_2, & y_2(0) = 1.0 \end{cases}$$

用变步长基尔方法计算当 $h=0.1$ 时，各积分点

$$t_i = i \times h, i = 0, 1, \cdots, 10$$

上的未知函数值 $y_{0i}, y_{1i}, y_{2i}(i=0,1,\cdots,10)$。取 $eps=0.000001$。

主函数程序及计算微分方程组右端函数值的函数示例程序如下：

```
//ODE_GillVariationalStep.cpp                变步长积分一步基尔法

#include <iostream>                          //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"    //求解常微分方程(组)头文件
using namespace std;                         //名字空间

void main(void)
{
    double q[3] = {0.0, 0.0, 0.0};
    double y[3] = {0.0, 1.0, 1.0};
    double t(0.0), h(0.1), eps(FLOATERROR);
    valarray<double> vy(y, 3), vq(q, 3);

    cout << "t = " << t << endl;
    for(int i = 0; i<3; i ++ )
      cout << " y(" << i << ") = " << y[i] << "\t";
    cout << endl;

    for(i = 1; i<11; i ++ )
    {
      ODE_GillVariationalStep(t,h,vy,eps,vq);
      t = h * i;
      cout << "t = " << t << endl;
      for(int j = 0; j<3; j ++ )
          cout << " y(" << j << ") = " << vy[j] << "\t";
      cout << endl;
  }
}
```

```
//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueGVS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();          //微分方程组中方程的个数,也是未知函数的个数

    d[0] = y[1];
    d[1] = -y[0];
    d[2] = -y[2];
}
```

8. 示例结果

```
t = 0.000000
    y(0) = 0.000000    y(1) = 1.000000    y(2) = 1.000000
t = 0.100000
    y(0) = 0.099833    y(1) = 0.995004    y(2) = 0.904837
t = 0.200000
    y(0) = 0.198669    y(1) = 0.980067    y(2) = 0.818731
t = 0.300000
    y(0) = 0.295520    y(1) = 0.955336    y(2) = 0.740818
t = 0.400000
    y(0) = 0.389418    y(1) = 0.921061    y(2) = 0.670320
t = 0.500000
    y(0) = 0.479426    y(1) = 0.877583    y(2) = 0.606531
t = 0.600000
    y(0) = 0.564642    y(1) = 0.825336    y(2) = 0.548812
t = 0.700000
    y(0) = 0.644218    y(1) = 0.764842    y(2) = 0.496585
t = 0.800000
    y(0) = 0.717356    y(1) = 0.696707    y(2) = 0.449329
t = 0.900000
    y(0) = 0.783327    y(1) = 0.621610    y(2) = 0.406570
t = 1.000000
    y(0) = 0.841471    y(1) = 0.540302    y(2) = 0.367879
```

本问题的解析解为

$$\begin{cases} y_0 = \sin t \\ y_1 = \cos t \\ y_2 = \mathrm{e}^{-t} \end{cases}$$

11.2.7 全区间变步长积分基尔法

1. 功能

用变步长基尔(Gill)公式对一阶微分方程组进行全区间积分。

2. 算法

同本章 11.2.6 节算法。

3. 函数原型

```
//全区间变步长积分基尔法
template <class _Ty>
void ODE_GillVariationalStepInterval(_Ty t, _Ty h, valarray<_Ty>& y,_Ty eps, int k, matrix<_Ty>& z);
```

4. 参数

t:_Ty 类型变量。积分的起始点。

h:_Ty 类型变量。积分步长。

y:valarray<_Ty>类型引用,长度为 n,存放起始点 t 处的 n 个未知函数值 $y_i(t)(i=0,1,\cdots,n-1)$;返回时其值在二维数组 z 的第零列,即

$$z(i,0)=y_i(t),i=0,1,\cdots,n-1$$

eps:_Ty 类型变量。积分精度要求。

k:整型变量。积分步长数(包括起始点这一步)。

z:matrix<_Ty>引用,体积为 $n\times k$。返回 k 个积分点上的 n 个未知数值,即

$$z(i,j)=y_i(t_j),i=0,1,\cdots,n-1;j=0,1,\cdots,k-1$$

其中 $z=(i,0)(i=0,1,\cdots,n-1)$为初值。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,\cdots,n-1)$的函数 FunctionValueRKCS(),由用户自己编写,其形式如下:

```
template <class _Ty>
void FunctionValueRKCS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();          //微分方程组中方程的个数,也是未知函数的个数
        ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
        ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y_0' = y_1, & y_0(0)=0.0 \\ y_1' = -y_0, & y_1(0)=1.0 \\ y_2' = -y_2, & y_2(0)=1.0 \end{cases}$$

用变步长基尔方法计算当 $h=0.1$ 时,各积分点

$$t_i=i\times h,i=0,1,\cdots,10$$

上的未知函数值 $y_{0i},y_{1i},y_{2i}(i=0,1,\cdots,10)$。取 $eps=0.000001$。

主函数程序及计算微分方程组右端函数值的函数示例程序如下:

```
//ODE_GillVariationalStepInterval.cpp                全区间变步长积分基尔法
```

```
#include <iostream>                                    //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"              //求解常微分方程(组)头文件
using namespace std;                                   //名字空间

void main(void)
{
    double y[3] = {0.0, 1.0, 1.0}, a(0.0), h(0.1), eps(FLOATERROR);
    valarray<double> vy(y, 3);
    matrix<double> vz(3,11);

    cout.setf(ios::fixed);                             //输出数据为定点法
    cout.precision(6);                                 //精度6位

    ODE_GillVariationalStepInterval(a,h,vy,eps,11,vz);

    for(int i=0; i<11; i++)
    {
      double t = h * i;
      cout << "t = " << t << endl;
      for(int j=0; j<3; j++)
          cout << " y(" << j << ") = " << vz(j, i) << "\t";
      cout << endl;
  }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueGVSI(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();        //微分方程组中方程的个数,也是未知函数的个数

    d[0] = y[1];
    d[1] = -y[0];
    d[2] = -y[2];
}
```

8. 示例结果

```
t = 0.000000
    y(0) = 0.000000    y(1) = 1.000000    y(2) = 1.000000
t = 0.100000
    y(0) = 0.099833    y(1) = 0.995004    y(2) = 0.904837
t = 0.200000
    y(0) = 0.198669    y(1) = 0.980067    y(2) = 0.818731
t = 0.300000
    y(0) = 0.295520    y(1) = 0.955336    y(2) = 0.740818
```

```
t = 0.400000
    y(0) = 0.389418     y(1) = 0.921061     y(2) = 0.670320
t = 0.500000
    y(0) = 0.479426     y(1) = 0.877583     y(2) = 0.606531
t = 0.600000
    y(0) = 0.564642     y(1) = 0.825336     y(2) = 0.548812
t = 0.700000
    y(0) = 0.644218     y(1) = 0.764842     y(2) = 0.496585
t = 0.800000
    y(0) = 0.717356     y(1) = 0.696707     y(2) = 0.449329
t = 0.900000
    y(0) = 0.783327     y(1) = 0.621610     y(2) = 0.406570
t = 1.000000
    y(0) = 0.841471     y(1) = 0.540302     y(2) = 0.367879
```

本问题的解析解为

$$\begin{cases} y_0 = \sin t \\ y_1 = \cos t \\ y_2 = \mathrm{e}^{-t} \end{cases}$$

11.2.8 变步长积分默森法

1. 功能

用变步长默森(Merson)法对一阶微分方程组进行全区间积分。

2. 算法

设一阶微分方程组为

$$\begin{cases} y_0 = f_0(t, y_0, y_1, \cdots, y_{n-1}), y(t_0) = y_{00} \\ y_1 = f_1(t, y_0, y_1, \cdots, y_{n-1}), y_1(t_0) = y_{10} \\ \quad \vdots \\ y_{n-1} = f_{n-1}(t, y_0, y_1, \cdots, y_{n-1}), y_{n-1}(t_0) = y_{n-1,0} \end{cases}$$

由 t_j 积分一步到点到 $t_{j+1}=t_j+h$ 点的默森法的计算公式为：

$$y_i^{(1)} = y_i^{(0)} + \frac{h}{3} f_i(t_j, y_0^{(0)}, y_1^{(0)}, \cdots, y_{n-1}^{(0)})$$

$$y_i^{(2)} = y_i^{(1)} + \frac{h}{6}\left[f_i\left(t_j + \frac{h}{3}, y_0^{(1)}, y_1^{(1)}, \cdots, y_{n-1}^{(1)}\right) - f_i(t_j, y_0^{(0)}, y_1^{(0)}, \cdots, y_{n-1}^{(0)}) \right]$$

$$y_i^{(3)} = y_i^{(2)} + \frac{3}{8}h\left[f_i\left(t_j + \frac{h}{3}, y_0^{(2)}, y_1^{(2)}, \cdots, y_{n-1}^{(2)}\right) - \frac{9}{4}\left(f_i\left(t_j + \frac{h}{3}, y_0^{(1)}, y_1^{(1)}, \cdots, y_{n-1}^{(1)}\right) + \frac{1}{4} f_i(t_j, y_0^{(0)}, \cdots, y_{n-1}^{(0)}) \right) \right]$$

$$y_i^{(4)} = y_i^{(3)} + 2h\left[f_i\left(t_j + \frac{h}{2}, y_0^{(3)}, y_1^{(3)}, \cdots, y_{n-1}^{(3)}\right) - \frac{15}{16}\left(f_i\left(t_j + \frac{h}{3}, y_0^{(2)}, y_1^{(2)}, \cdots, y_{n-1}^{(2)}\right) - \frac{1}{5} f_i(t_j, y_0^{(0)}, y_1^{(0)}, \cdots, y_{n-1}^{(0)}) \right) \right]$$

$$y_i^{(5)} = y_i^{(4)} + \frac{h}{6}\left[f_i(t_j + h, y_0^{(4)}, y_1^{(4)}, \cdots, y_{n-1}^{(4)}) - 8\left(f_i\left(t_j + \frac{h}{2}, y_0^{(3)}, y_1^{(3)}, \cdots, y_{n-1}^{(3)}\right) \right.\right.$$

$$-\frac{9}{8}\left(f_i\left(t_j+\frac{h}{3},y_0^{(2)},y_1^{(2)},\cdots,y_{n-1}^{(2)}\right)-\frac{2}{9}f_i(t_j,y_0^{(0)},y_1^{(0)},\cdots,y_{n-1}^{(0)})\right)\Big]$$

$$i=0,1,\cdots,n-1$$

其中

$$y_i^{(0)}=y_i(t_j),i=0,1,\cdots,n-1$$

$$y_i^{(5)}=y_i(t_j+h),i=0,1,\cdots,n-1$$

关于自动选择步长参看本章 11.2.5 节的算法。

3. 函数原型

```
//变步长积分默森法
template <class _Ty>
void ODE_MersonVariationalStep(_Ty t, _Ty h, int n, valarray<_Ty>& y,_Ty eps, int k, matrix<_Ty>& z);
```

4. 参数

t:_Ty 类型变量。积分的起始点。

h:_Ty 类型变量。积分一步的步长。

y:valarray<_Ty>类型引用,长度为 n,存放起始点 t 处的 n 个未知函数的初值 $y_i(t)(i=0,1,\cdots,n-1)$;返回时其初值在二维数组 z 的第零列中,即

$$z(i,0)=y_i(t),(i=0,1,\cdots,n-1)$$

eps:_Ty 类型变量。控制积分一步的精度要求。

k:整型变量。积分步数(包括起始点这一步)。

z:matrix<_Ty>引用,体积为 $n\times k$。返回微分方程组中 n 个未知函数在 k 个积分点(包括起始点)处的函数值,即

$$z(i,j)=y_{ij},\quad i=0,1,\cdots,n-1;\quad j=0,1,\cdots,k-1$$

其中 $z(i,0)(i=0,1,\cdots,n-1)$为初值。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,\cdots,n-1)$的函数 FunctionValueMVS(),由用户自己编写,其形式如下:

```
template <class _Ty>
void FunctionValueMVS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();     //微分方程组中方程的个数,也是未知函数的个数
        ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
        ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y_0' = 60[0.06 + t(t-0.6)]y_1, & y_0(0) = 0.0 \\ y_1' = -60[0.06 + t(t-0.6)]y_0, & y_1(0) = 1.0 \end{cases}$$

用默森方法计算当 $h=0.1$ 时，各积分点 $t_i = i \times h, i = 0, 1, \cdots, 10$，上的未知函数值 $y_0(t_i)$ 及 $y_1(t_i)(i=0,1,\cdots,10)$ 的近似值。取 $eps=0.00001$。

主函数程序及计算微分方程组右端函数值的函数示例程序如下：

```
//ODE_MersonVariationalStep.cpp                    变步长积分默森法

#include <iostream>                                //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"          //求解常微分方程(组)头文件
using namespace std;                               //名字空间

void main(void)
{
    double y[2] = {0.0, 1.0}, t(0.0), h(0.1), eps(FLOATERROR);
    valarray<double> vy(y, 2);
    matrix<double> vz(2,11);

    cout.setf(ios::fixed);                         //输出数据为定点法
    cout.precision(6);                             //精度 6 位

    ODE_MersonVariationalStep(t,h,2,vy,eps,11,vz);

    for(int i = 0; i<11; i++)
    {
      double x = h * i;
      cout << "t = " << x << endl;
      cout << " y(0) = " << vz(0, i) << "\t y(1) = " << vz(1, i) << endl;
   }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueMVS(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    _Ty q;

    int n = y.size();        //微分方程组中方程的个数,也是未知函数的个数

    q = 60.0 * (0.06 + t* (t - 0.6));
    d[0] = q * y[1];
    d[1] = -q * y[0];
}
```

8. 示例结果

```
t = 0.000000
```

```
    y(0) = 0.000000      y(1) = 1.000000
t = 0.100000
    y(0) = 0.198669      y(1) = 0.980066
t = 0.200000
    y(0) = 0.159318      y(1) = 0.987227
t = 0.300000
    y(0) = -0.000000     y(1) = 1.000000
t = 0.400000
    y(0) = -0.159318     y(1) = 0.987227
t = 0.500000
    y(0) = -0.198669     y(1) = 0.980066
t = 0.600000
    y(0) = -0.000000     y(1) = 1.000000
t = 0.700000
    y(0) = 0.531186      y(1) = 0.847255
t = 0.800000
    y(0) = 0.999574      y(1) = -0.029200
t = 0.900000
    y(0) = -0.098249     y(1) = -0.995162
t = 1.000000
    y(0) = -0.631266     y(1) = 0.775566
```

本问题的解析解为

$$\begin{cases} y_0 = \sin[20t(t-0.3)(t-0.6)] \\ y_1 = \cos[20t(t-0.3)(t-0.6)] \end{cases}$$

11.2.9 积分一步连分式法

1. 功能

用连分式法对一阶微分方程组积分一步。

2. 算法

设一阶微分方程组为

$$\begin{cases} y_0 = f_0(t, y_0, y_1, \cdots, y_{n-1}), y(t_0) = y_{00} \\ y_1 = f_1(t, y_0, y_1, \cdots, y_{n-1}), y_1(t_0) = y_{10} \\ \quad \vdots \\ y_{n-1} = f_{n-1}(t, y_0, y_1, \cdots, y_{n-1}), y_{n-1}(t_0) = y_{n-1,0} \end{cases}$$

由 t_j 积分一步到点到 $t_{j+1}=t_j+h$ 点的有理分式如下。

用定步长四阶龙格一库塔(Runge-Kutta)法由 t_j 跨 $n=2^k$ 步到 t_{j+1}，其中每一小步的步长为

$$h_k = h/2^k, k = 0, 1, \cdots$$

由此可以计算得到

$$y_{i,j+1}^{(k)} \xrightarrow{\text{记作}} s_{ki}, i = 0, 1, \cdots, n-1$$

显然，在 t_{j+1} 处的各未知数的近似值是步长 h^* 的函数，设为 $s_i(h^*)$ 即

$$y_{i,j+1} = y_i(t_{j+1}) \approx s_i(h^*)$$

并且，当 h^* 越小时，计算得到的 $s_i(h^*)$ 越接近于准确值 $y_i(t_{j+1})$。

若 $s_i(h^*)$ 用连分式表示为

$$s_i(h^*) = b_{0i} + \frac{h^* - h_0}{b_{1i}} + \frac{h^* - h_1}{b_{2i}} + \cdots + \frac{h^* - h_{l-1}}{b_{li} + \cdots}$$

$$i = 0,1,\cdots,n-1$$

则有

$$y_i(t_{j+1}) = s_i(0)$$

$$= b_{0i} - \frac{h_0}{b_{1i}} - \frac{h_1}{b_{2i}} - \cdots - \frac{h_{l-1}}{b_{li} - \cdots} \quad i = 0,1,\cdots,n-1$$

其中，$b_{li}(l=0,1,\cdots)$ 可由以下递推公式计算：

$$b_{0i} = s_{0i}$$

$$\begin{cases} d_{0i} = s_{li} \\ d_{ki} = \dfrac{h_l - h_{k-1}}{d_{k-1,i} - b_{k-1,i}}, k = 1,2,\cdots,l \end{cases}$$

$$b_{li} = d_{li}$$

其中

$$i = 0,1,\cdots,n-1$$

通常，以上连分式作到七节就能满足精度要求。下面是用连分式法由 t_j 积分一步到 $t_{j+1} = t_j + h$ 的计算步骤。

(1) 用定步长四阶龙格－库塔法公式由 t_j 跨一步到 t_{j+1} 计算 $s_{0i}(i = 0,1,\cdots,n-1)$。其中

$$h_0 = t_{j+1} - t_j = h$$

$$b_{0i} = s_{0i}, y_i^{(0)} = s_{0i}, i = 0,1,\cdots,n-1$$

(2) 对于 $l = 1,2\cdots$，令

$$h_l = h_{l-1}/2$$

以 h_l 为步长，用定步长四阶龙格－库塔法公式由 t_j 跨 $n = 2^l$ 步到 t_{j+1} 计算 $s_{li}(i = 0,1,\cdots,n-1)$。然后由下列递推公式计算 $b_{li}(i = 0,1,\cdots,n-1)$：

$$d_i = s_{li}$$

$$d_i = \frac{h_l - h_k}{d_i - b_{ki}}, k = 0,1,\cdots,l-1$$

$$b_{ki} = d_i$$

最后计算

$$y_i^{(l)} = b_{0i} - \frac{h_0}{b_{1i}} - \frac{h_1}{b_{2i}} - \cdots - \frac{h_{l-1}}{b_{li}}, i = 0,1,\cdots,n-1$$

反复作步骤(2)直到 $\max\limits_{0 \leqslant i \leqslant n-1} |y_i^{(l)} - y_i^{(l-1)}| < \varepsilon$ 为止。

3. 函数原型

```
//积分一步连分式法
template <class _Ty>
void ODE_Fraction(_Ty t, _Ty h, valarray<_Ty>& y, _Ty eps);
```

4. 参数

t:_Ty 类型变量。积分的起始点。

h:_Ty 类型变量。积分一步的步长。

y:valarray<_Ty>类型引用，长度为 n，存放 n 个未知函数在起始点 t 处的函数值 $y_i(t)(i=0,1,\cdots,n-1)$；返回时存放 n 个未知数在其初值在 $t+h$ 点的函数值 $y_i(t+h)(i=0,1,\cdots,n-1)$。

eps:_Ty 类型变量。积分一步的精度要求。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,\cdots,n-1)$ 的函数 FunctionValueF()，由用户自己编写，其形式如下：

```
template <class _Ty>
void FunctionValueF(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();          //微分方程组中方程的个数，也是未知函数的个数
      ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
      ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y'_0 = -y_1, & y_0(0)=1.0 \\ y'_1 = y_0, & y_1(0)=0.0 \end{cases}$$

用连分式法计算当 $h=0.1$ 时，各积分点

$$t_i = i \times h, i = 0,1,\cdots,10$$

上的未知函数值 $y_i(t_j)(i=0,1;j=0,1,\cdots,10)$。取 $eps=0.000001$。

主函数程序及计算微分方程组右端函数值的函数示例程序如下：

```
//ODE_Fraction.cpp                          积分一步连分式法

#include <iostream>                         //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"   //求解常微分方程(组)头文件
using namespace std;                        //名字空间

void main(void)
{
    double dy[2] = {1.0, 0.0}, t(0.0), h(0.1), eps(FLOATERROR);
    valarray<double> y(dy, 2);

    cout.setf(ios::fixed);          //输出数据为定点法
```

```
    cout.precision(6);                    //精度6位

    cout << " t = " << t << "\t y[0] = " << dy[0] << "\t y[1] = " << dy[1] << endl;

    for(int i = 0; i<10; i++)
    {
      ODE_Fraction(t, h, y, eps);
      t = t + h;
      cout << " t = " << t << "\t y[0] = " << y[0] << "\t y[1] = " << y[1] << endl;
    }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueF(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
      int n = y.size();      //微分方程组中方程的个数,也是未知函数的个数

      d[0] = -y[1];
      d[1] = y[0];
}
```

8. 示例结果

```
t = 0.000000  y[0] = 1.000000  y[1] = 0.000000
t = 0.100000  y[0] = 0.995004  y[1] = 0.099833
t = 0.200000  y[0] = 0.980067  y[1] = 0.198669
t = 0.300000  y[0] = 0.955336  y[1] = 0.295520
t = 0.400000  y[0] = 0.921061  y[1] = 0.389419
t = 0.500000  y[0] = 0.877582  y[1] = 0.479426
t = 0.600000  y[0] = 0.825335  y[1] = 0.564643
t = 0.700000  y[0] = 0.764842  y[1] = 0.644218
t = 0.800000  y[0] = 0.696706  y[1] = 0.717357
t = 0.900000  y[0] = 0.621609  y[1] = 0.783327
t = 1.000000  y[0] = 0.540302  y[1] = 0.841471
```

本问题的解析解为

$$\begin{cases} y_0 = \cos t \\ y_1 = \sin t \end{cases}$$

11.2.10 全区间积分连分式法

1. 功能

用连分式法对一阶微分方程组进行全区间积分。

2. 算法

同本章11.2.9节算法。

3. 函数原型

```
//全区间积分连分式法
template <class _Ty>
void ODE_FractionInterval(_Ty t, _Ty h, valarray<_Ty>& y,_Ty eps, int k, matrix<_Ty>& z);
```

4. 参数

t:_Ty 类型变量。积分的起始点。

h:_Ty 类型变量。积分一步的步长。

y:valarray<_Ty>类型引用，长度为 n，存放起始点 t 处的 n 个未知函数值(即初值)$y_i(t)(i=0,1,\cdots,n-1)$;返回时其初值在二维数组 z 的第零列中，即

$$z(i,0)=y_i(t),(i=0,1,\cdots,n-1)$$

eps:_Ty 类型变量。积分一步的精度要求。

k:整型变量。积分的步数(包括起始点这一步)。

z:matrix<_Ty>引用，体积为 $n\times k$。返回 n 个未知函数在 k 个积分点上的函数值，即

$$z(i,j)=y_i(t_j),i=0,1,\cdots,n-1;j=0,1,\cdots,k-1$$

其中 $z(i,0)=y_i(t)(i=0,1,\cdots,n-1)$。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,\cdots,n-1)$ 的函数 FunctionValueF()，由用户自己编写，其形式如下：

```
template <class _Ty>
void FunctionValueF(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();      //微分方程组中方程的个数,也是未知函数的个数
        ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
        ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases}y'_0=-y_1, & y_0(0)=1.0\\ y'_1=y_0, & y_1(0)=0.0\end{cases}$$

用全区间积分的连分式法计算当 $h=0.1$ 时，各积分点

$$t_i=i\times h,i=0,1,\cdots,10$$

上的未知函数值 $y_0(t_i)$ 及 $y_1(t_j)(i=0,1;j=0,1\cdots,10)$。取 $eps=0.000001$。

主函数程序及计算微分方程组右端函数值的函数示例程序如下：

```
//ODE_FractionInterval.cpp                    全区间积分连分式法
```

```
#include <iostream>                          //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"   //求解常微分方程(组)头文件
using namespace std;                         //名字空间

void main(void)
{
    double y[2] = {1.0, 0.0}, t(0.0), h(0.1), eps(FLOATERROR);
    valarray<double> vy(y, 2);
    matrix<double> vz(2,11);

    cout.setf(ios::fixed);              //输出数据为定点法
    cout.precision(6);                  //精度6位

    ODE_FractionInterval(t,h,vy,eps,11,vz);

    for(int i=0; i<11; i++)
    {
      double t = h * i;
      cout << "t = " << t << endl;
      for(int j=0; j<2; j++)
          cout << " y(" << j << ") = " << vz(j, i) << "\t";
      cout << endl;
    }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueFI(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
      int n = y.size();      //微分方程组中方程的个数,也是未知函数的个数

      d[0] = -y[1];
      d[1] = y[0];
}
```

8. 示例结果

```
t = 0.000000
    y(0) = 1.000000   y(1) = 0.000000
t = 0.100000
    y(0) = 0.995004   y(1) = 0.099833
t = 0.200000
    y(0) = 0.980067   y(1) = 0.198669
t = 0.300000
    y(0) = 0.955336   y(1) = 0.295520
t = 0.400000
    y(0) = 0.921061   y(1) = 0.389419
t = 0.500000
```

```
    y(0) = 0.877582  y(1) = 0.479426
t = 0.600000
    y(0) = 0.825335  y(1) = 0.564643
t = 0.700000
    y(0) = 0.764842  y(1) = 0.644218
t = 0.800000
    y(0) = 0.696706  y(1) = 0.717357
t = 0.900000
    y(0) = 0.621609  y(1) = 0.783327
t = 1.000000
    y(0) = 0.540302  y(1) = 0.841471
```

本问题的解析解为

$$\begin{cases} y_0 = \cos t \\ y_1 = \sin t \end{cases}$$

11.2.11 全区间积分双边法

1. 功能

用双边法对一阶微分方程组进行全区间积分。

2. 算法

设一阶微分方程组为

$$\begin{cases} y'_0 = f_0(t, y_0, y_1, \cdots, y_{n-1}), y(t_0) = y_{00} \\ y'_1 = f_1(t, y_0, y_1, \cdots, y_{n-1}), y_1(t_0) = y_{10} \\ \quad \vdots \\ y'_{n-1} = f_{n-1}(t, y_0, y_1, \cdots, y_{n-1}), y_{n-1}(t_0) = y_{n-1,0} \end{cases}$$

用双边法积分一步(步长为 h)的计算公式为:

$$p_i^{(j+2)} = -4y_i^{(j+1)} + 5y_i^{(j)} + 2h[2f_i^{(j+1)} + f_i^{(j)}]$$

$$q_i^{(j+2)} = 4y_i^{(j+1)} - 3y_i^{(j)} + \frac{2}{3}h[f_i^{(j+2)} - 2f_i^{(j+1)} - 2f_i^{(j)}]$$

$$y_i^{(j+2)} = \frac{1}{2}[p_i^{(j+2)} + q_i^{(j+2)}] \quad i = 0, 1, \cdots, n-1$$

其中

$$y_i^{(j)} = y_i(t_j)$$

$$f_i^{(j)} = f_i(t_j, y_0^{(j)}, y_1^{(j)}, \cdots, y_{n-1}^{(j)})$$

$$f_i^{(j+1)} = f_i(t_j + h, y_0^{(j+1)}, y_1^{(j+1)}, \cdots, y_{n-1}^{(j+1)})$$

$$f_i^{(j+2)} = f_i(t_j + 2h, p_0^{(j+2)}, p_1^{(j+2)}, \cdots, p_{n-1}^{(j+2)}) \quad i = 0, 1, \cdots, n-1$$

本方法为多步法,在进行全区间积分时,要求采用某种单步法起步算出 $y_i(t_1)(i = 0, 1, \cdots, n-1)$。在本函数中,采用积分一步的变步长龙格－库塔(Runge-Kutta)法起步算出 $y_i(t_1)(i = 0, 1, \cdots, n-1)$。

3. 函数原型

```
//全区间积分双边法
template <class _Ty>
```

```
void ODE_BothSidesInterval(_Ty t, _Ty h, valarray<_Ty>& y,_Ty eps, int k, matrix<_Ty>& z);
```

4. 参数

t:_Ty 类型变量。积分起始点 t_0 。

h:_Ty 类型变量。积分一步的步长。

y:valarray<_Ty>类型引用，长度为 n。存放在起始点 t 处的 n 个未知函数值(即初值)$y_i(t)(i=0,1,\cdots,n-1)$。返回时其值在二维数组 z 的第零列中，即

$$z(i,0)=y_i(t),i=0,1,\cdots,n-1$$

eps:_Ty 类型变量。在调用变步长龙格－库塔法函数时的控制精度要求。

k:整型变量。积分的步数(包括起始点这一步)。

z:matrix<_Ty>引用，体积为 $n\times k$。返回 k 个积分点(包括起始点)处 n 个未知函数值，即

$$z(i,j)=y_i(t_j),i=0,1,\cdots,n-1;j=0,1,\cdots,k-1$$

其中 $z(i,0)=y_i(t)$ 为初值。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,\cdots,n-1)$ 的函数 FunctionValueBSI()，由用户自己编写，其形式如下：

```
template <class _Ty>
void FunctionValueBSI(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();      //微分方程组中方程的个数，也是未知函数的个数
        ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
        ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y'_0=-y_1, & y_0(0)=1.0 \\ y'_1=y_0, & y_1(0)=0.0 \end{cases}$$

用双边法计算当 $h=0.1$ 时，各积分点

$$t_i=i\times h,i=0,1,\cdots,10$$

处的未知函数 $y_0(t_i)$ 及 $y_1(t_i)(i=0,1,\cdots,10)$。取 $eps=0.00001$。

主函数程序及计算微分方程组右端函数值的函数示例程序如下：

```
//ODE_BothSidesInterval.cpp                    全区间积分连分式法

#include <iostream>                            //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"      //求解常微分方程(组)头文件
using namespace std;                           //名字空间
```

```
void main(void)
{
    double y[2] = {1.0, 0.0}, t(0.0), h(0.1), eps(FLOATERROR);
    valarray<double> vy(y, 2);
    matrix<double> vz(2,11);

    cout.setf(ios::fixed);          //输出数据为定点法
    cout.precision(6);              //精度6位

    ODE_BothSidesInterval(t,h,vy,eps,11,vz);

    for(int i=0; i<11; i++)
    {
    double x = h * i;
    cout << "t = " << x << endl;
    for(int j=0; j<2; j++)
        cout << " y(" << j << ") = " << vz(j, i) << "\t";
    cout << endl;
    }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueBSI(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{

    int n = y.size();     //微分方程组中方程的个数,也是未知函数的个数

    d[0] = -y[1];
    d[1] = y[0];
}
```

8. 示例结果

```
t = 0.000000
    y(0) = 1.000000  y(1) = 0.000000
t = 0.100000
    y(0) = 0.995004  y(1) = 0.099833
t = 0.200000
    y(0) = 0.980067  y(1) = 0.198669
t = 0.300000
    y(0) = 0.955337  y(1) = 0.295520
t = 0.400000
    y(0) = 0.921061  y(1) = 0.389417
t = 0.500000
    y(0) = 0.877583  y(1) = 0.479425
```

```
t = 0.600000
    y(0) = 0.825336  y(1) = 0.564641
t = 0.700000
    y(0) = 0.764843  y(1) = 0.644217
t = 0.800000
    y(0) = 0.696708  y(1) = 0.717355
t = 0.900000
    y(0) = 0.621611  y(1) = 0.783326
t = 1.000000
    y(0) = 0.540304  y(1) = 0.841470
```

本问题的解析解为

$$\begin{cases} y_0 = \cos t \\ y_1 = \sin t \end{cases}$$

11.2.12 全区间积分阿当姆斯预报一校正法

1. 功能

用阿当姆斯(Adams)预报－校正公式对一阶微分方程组进行全区间积分。

2. 算法

设一阶微分方程组为

$$\begin{cases} y'_0 = f_0(t, y_0, y_1, \cdots, y_{n-1}), y(t_0) = y_{00} \\ y'_1 = f_1(t, y_0, y_1, \cdots, y_{n-1}), y_1(t_0) = y_{10} \\ \quad\vdots \\ y'_{n-1} = f_{n-1}(t, y_0, y_1, \cdots, y_{n-1}), y_{n-1}(t_0) = y_{n-1,0} \end{cases}$$

积分步长为 h，则阿当姆斯预报－校正公式为

预报公式 $\bar{y}_{i,j+1} = y_{ij} + [55f_{ij} - 59f_{i,j-1} + 37f_{i,j-2} - 9f_{i,j-3}]h/24$

校正公式 $y_{i,j+1} = y_{ij} + [9f_{i,j+1} + 19f_{ij} - 5f_{i,j-1} + f_{i,j-2}]h/24$

$$i = 0, 1, \cdots, n-1$$

其中

$$f_{ik} = f_i(t_k, y_{0k}, y_{1k}, \cdots, y_{n-1,k})$$

$$k = j-3, j-2, j-1, j; i = 0, 1, \cdots, n-1$$

$$f_{i,j+1} = f_i(t_{j+1}, \bar{y}_{0,j+1}, \bar{y}_{1,j+1}, \cdots, \bar{y}_{n-1,j+1}), i = 0, 1, \cdots, n-1$$

阿当姆斯方法是线性多步法，在计算新点上的未知函数值时要用到前面 4 个点的未知函数值。在本函数中，采用变步长四阶龙格－库塔公式计算开始 4 个点上的未知数值 $y_{i0}, y_{i1}, y_{i2}, y_{i3}(i = 0, 1, \cdots, n-1)$，其中 y_{i0} 为给定的初值。

3. 函数原型

```
//全区间积分阿当姆斯预报－校正法
template <class _Ty>
void ODE_AdamsInterval(_Ty t, _Ty h, valarray<_Ty>& y,_Ty eps, int k, matrix<_Ty>& z);
```

4. 参数

t：_Ty 类型变量。积分的起始点。

h：_Ty 类型变量。积分一步的步长。

y:valarray<_Ty>类型引用，长度为 n。存放在起始点 t 处的 n 个未知函数值（即初值）$y_i(t)(i=0,1,\cdots,n-1)$。返回时其值在二维数组 z 的第零列，即

$$z(i,0)=y_i(t),i=0,1,\cdots,n-1$$

eps:_Ty 类型变量。在调用变步长龙格－库塔法函数时的控制精度要求。

k:整型变量。积分的步数（包括起始点这一步）。

z:matrix<_Ty>引用，体积为 $n\times k$。返回 k 个积分点（包括起始点）处 n 个未知函数值 $y_i(t_j)(i=0,1,\cdots,n-1;j=0,1,\cdots,k-1)$，即

$$z(i,j)=y_i(t_j),i=0,1,\cdots,n-1;j=0,1,\cdots,k-1$$

其中 $z(i,0)=y_i(t)$ 为初值。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,\cdots,n-1)$ 的函数 FunctionValueAI()，由用户自己编写，其形式如下：

```
template <class _Ty>
void FunctionValueAI(_Ty t, valarray<_Ty>& y,valarray<_Ty>& d)
{
    int n = y.size();          //微分方程组中方程的个数，也是未知函数的个数
      ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
      ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y'_0 = y_1, & y_0(0)=0.0 \\ y'_1 = -y_0, & y_1(0)=1.0 \\ y'_2 = -y_2, & y_2(0)=1.0 \end{cases}$$

用阿当预报－校正公式计算当 $h=0.05$ 时，各积分点

$$t_i = i\times h,\ i=0,1,\cdots,10$$

上的未知函数 $y_{0i},y_{1i},y_{2i}(i=0,1,\cdots,10)$。取 $eps=10^{-6}$。

主函数程序及计算微分方程组右端函数值的函数示例程序如下：

```
//ODE_AdamsInterval.cpp      全区间积分阿当姆斯预报－校正法

#include <iostream>                              //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"        //求解常微分方程(组)头文件
using namespace std;                             //名字空间

void main(void)
```

```
{
    double y[3] = {0.0, 1.0, 1.0}, t(0.0), h(0.05), eps(FLOATERROR);
    valarray<double> vy(y, 3);
    matrix<double> vz(3,11);

    cout.setf(ios::fixed);              //输出数据为定点法
    cout.precision(6);                  //精度 6 位

    ODE_AdamsInterval(t,h,vy,eps,11,vz);

    for(int i = 0; i<11; i++)
    {
    double t = h * i;
    cout << "t = " << t << endl;
    for(int j = 0; j<3; j++)
        cout << " y(" << j << ") = " << vz(j, i) << "\t";
    cout << endl;
  }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueAI(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();                   //微分方程组中方程的个数,也是未知函数的个数

    d[0] = y[1];
    d[1] = -y[0];
    d[2] = -y[2];
}
```

8. 示例结果

```
t = 0.000000
    y(0) = 0.000000  y(1) = 1.000000  y(2) = 1.000000
t = 0.050000
    y(0) = 0.049979  y(1) = 0.998750  y(2) = 0.951229
t = 0.100000
    y(0) = 0.099833  y(1) = 0.995004  y(2) = 0.904837
t = 0.150000
    y(0) = 0.149438  y(1) = 0.988771  y(2) = 0.860708
t = 0.200000
    y(0) = 0.198669  y(1) = 0.980067  y(2) = 0.818731
t = 0.250000
    y(0) = 0.247404  y(1) = 0.968912  y(2) = 0.778801
t = 0.300000
```

```
    y(0) = 0.295520  y(1) = 0.955336  y(2) = 0.740818
t = 0.350000
    y(0) = 0.342898  y(1) = 0.939373  y(2) = 0.704688
t = 0.400000
    y(0) = 0.389418  y(1) = 0.921061  y(2) = 0.670320
t = 0.450000
    y(0) = 0.434966  y(1) = 0.900447  y(2) = 0.637628
t = 0.500000
    y(0) = 0.479426  y(1) = 0.877583  y(2) = 0.606531
```

本问题的解析解为

$$\begin{cases} y_0 = \sin t \\ y_1 = \cos t \\ y_2 = \mathrm{e}^{-t} \end{cases}$$

11.2.13 全区间积分哈明法

1. 功能

用哈明(Hamming)方法对一阶微分方程组进行全区间积分。

2. 算法

设一阶微分方程组为

$$\begin{cases} y'_0 = f_0(t, y_0, y_1, \cdots, y_{n-1}), y(t_0) = y_{00} \\ y'_1 = f_1(t, y_0, y_1, \cdots, y_{n-1}), y_1(t_0) = y_{10} \\ \quad \vdots \\ y'_{n-1} = f_{n-1}(t, y_0, y_1, \cdots, y_{n-1}), y_{n-1}(t_0) = y_{n-1,0} \end{cases}$$

积分步长为 h，则哈明方法的计算公式为：

预报 $\overline{P}_{i,j+1} = \overline{y}_{i,j-3} + 4h(2f_{ij} - f_{i,j-1} + 2f_{i,j-2})/3$

修正 $P_{i,j+1} = \overline{P}_{i,j+1} + \dfrac{112}{121}(C_{ij} - P_{ij})$

校正 $C_{i,j+1} = \dfrac{1}{8}[9y_{ij} - y_{i,j-2} + 3h(f_{i,j+1} + 2f_{ij} - f_{i,j-1})]$

$$i = 0, 1, \cdots, n-1$$

其中

$$f_{ik} = f_i(t_k, y_{0k}, y_{1k}, \cdots, y_{n-1,k}), \quad k = j-2, j-1, j; \quad i = 0, 1, \cdots, n-1$$

$$f_{i,j+1} = f_i(t_{j+1}, P_{0,j+1}, P_{1,j+1}, \cdots, P_{n-1,j+1}), i = 0, 1, \cdots, n-1$$

终值 $y_{i,j+1} = C_{i,j+1} - \dfrac{9}{121}(C_{i,j+1} - P_{i,j+1}), i = 0, 1, \cdots, n-1$

哈明方法是线性多步法，在计算新点上的未知函数值时要用到前面 4 个点上的未知函数值。在本函数中，采用积分一步的变步长龙格－库塔法计算开始 4 个点上的未知函数值 y_{i0}，$y_{i1}, y_{i2}, y_{i3} (i = 0, 1, \cdots, n-1)$，其中 y_{i0} 为初值 。

3. 函数原型

```
//全区间积分哈明法
template <class _Ty>
void ODE_HammingInterval(_Ty t, _Ty h, valarray<_Ty>& y,
```

```
_Ty eps, int k, matrix<_Ty>& z);
```

4. 参数

t:_Ty 类型变量。积分起始点。

h:_Ty 类型变量。积分一步的步长。

y:valarray < _Ty > 类型引用，长度为 n。存放在起始点 t 处的 n 个未知函数的初值 $y_i(t)(i=0,1,\cdots,n-1)$。返回时其值在二维数组 z 的第零列，即

$$z(i,0)=y_i(t),i=0,1,\cdots,n-1$$

eps:_Ty 类型变量。在调用积分一步的变步长龙格－库塔法时的精度要求。

k:整型变量。积分的步数(包括起始点这一步)。

z:matrix<_Ty>引用，体积为 $n\times k$。返回 k 个积分点(包括起始点)上 n 个未知函数值 $y_i(t_j)(i=0,1,\cdots,n-1;j=0,1,\cdots,k-1)$，即

$$z(i,j)=y_i(t_j),i=0,1,\cdots,n-1;j=0,1,\cdots,k-1$$

其中 $z(i,0)$ 为初值。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,\cdots,n-1)$ 的函数 FunctionValueAI()，由用户自己编写，其形式如下：

```
template <class _Ty>
void FunctionValueAI(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();          //微分方程组中方程的个数，也是未知函数的个数
      ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
      ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y'_0 = y_1, & y_0(0)=1.0 \\ y'_1 = -y_0, & y_1(0)=1.0 \\ y'_2 = -y_2, & y_2(0)=1.0 \end{cases}$$

用阿当预报－校正公式计算当 $h=0.05$ 时，各积分点

$$t_i=i\times h,i=0,1,\cdots,10$$

上的未知函数 $y_{0i},y_{1i},y_{2i}(i=0,1,\cdots,10)$。取 $eps=0.0001$。

主函数程序及计算微分方程组右端函数值的函数示例程序如下：

```
//ODE_HammingInterval.cpp                    全区间积分哈明法

#include <iostream>                          //输入输出流头文件
```

```
#include "OrdinaryDifferentialEguation.h"   //求解常微分方程(组)头文件
using namespace std;                        //名字空间

void main(void)
{
    double y[3] = {1.0, 1.0, 1.0}, t(0.0), h(0.05), eps(FLOATERROR);
    valarray<double> vy(y, 3);
    matrix<double> vz(3,11);

    cout.setf(ios::fixed);          //输出数据为定点法
    cout.precision(6);              //精度6位

    ODE_HammingInterval(t, h, vy, eps, 11, vz);

    for(int i=0; i<11; i++)
    {
      double x = h * i;
      cout << "t = " << x << endl;
      for(int j=0; j<3; j++)
          cout << " y(" << j << ") = " << vz(j, i) << "\t";
      cout << endl;
  }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueHI(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
     int n = y.size();      //微分方程组中方程的个数,也是未知函数的个数

     d[0] = y[1];
     d[1] = -y[0];
     d[2] = y[2];
}
```

8. 示例结果

```
t =0.000000
   y(0) = 1.000000    y(1) = 1.000000    y(2) = 1.000000
t = 0.050000
   y(0) = 1.048729    y(1) = 0.948771    y(2) = 1.051271
t = 0.100000
   y(0) = 1.094838    y(1) = 0.895171    y(2) = 1.105171
t = 0.150000
   y(0) = 1.138209    y(1) = 0.839333    y(2) = 1.161834
t = 0.200000
```

```
    y(0) = 1.178736     y(1) = 0.781397     y(2) = 1.221403
t = 0.250000
    y(0) = 1.216316     y(1) = 0.721508     y(2) = 1.284025
t = 0.300000
    y(0) = 1.250857     y(1) = 0.659816     y(2) = 1.349859
t = 0.350000
    y(0) = 1.282271     y(1) = 0.596475     y(2) = 1.419068
t = 0.400000
    y(0) = 1.310479     y(1) = 0.531643     y(2) = 1.491825
t = 0.450000
    y(0) = 1.335413     y(1) = 0.465482     y(2) = 1.568312
t = 0.500000
    y(0) = 1.357008     y(1) = 0.398157     y(2) = 1.648721
```

本问题的解析解为

$$\begin{cases} y_0 = \sin t + \cos t \\ y_1 = \cos t - \sin t \\ y_2 = e^t \end{cases}$$

11.2.14 积分一步特雷纳法

1. 功能

用特雷纳(Treanor)方法对一阶刚性(stiff)微分方程组积分一步。

2. 算法

设一阶微分方程组为

$$\begin{cases} y'_0 = f_0(t, y_0, y_1, \cdots, y_{n-1}), y(t_0) = y_{00} \\ y'_1 = f_1(t, y_0, y_1, \cdots, y_{n-1}), y_1(t_0) = y_{10} \\ \quad \vdots \\ y'_{n-1} = f_{n-1}(t, y_0, y_1, \cdots, y_{n-1}), y_{n-1}(t_0) = y_{n-1,0} \end{cases}$$

如果矩阵

$$\begin{bmatrix} \dfrac{\partial f_0}{\partial y_0} & \dfrac{\partial f_0}{\partial y_1} & \cdots & \dfrac{\partial f_0}{\partial y_{n-1}} \\ \dfrac{\partial f_1}{\partial y_0} & \dfrac{\partial f_1}{\partial y_1} & \cdots & \dfrac{\partial f_1}{\partial y_{n-1}} \\ \vdots & \vdots & & \vdots \\ \dfrac{\partial f_{n-1}}{\partial y_0} & \dfrac{\partial f_{n-1}}{\partial y_1} & & \dfrac{\partial f_{n-1}}{\partial y_{n-1}} \end{bmatrix}$$

的特征值 λ_i 具有如下特性：

$$\mathrm{Re}\lambda_i < 0 \text{ 且 } \max_{0 \leqslant i \leqslant n-1} |\mathrm{Re}\lambda_i| \gg \min_{0 \leqslant i \leqslant n-1} |\mathrm{Re}\lambda_i|$$

则称此微分方程组是刚性的。

求解刚性方程的特雷纳方法如下。

设已知 t_j 点处的未知函数值 $y_{i,j}(i = 0, 1, \cdots, n-1)$，则计算 $t_{j+1} = t_j + h$ 点处未知函数值 $y_{i,j+1}$ 的公式为

$$y_{i,j+1}=y_{ij}+\Delta y_i,\quad i=0,1,\cdots,n-1$$

其中

$$\Delta y_i=\begin{cases}\dfrac{h}{6}[q_i^{(1)}+2(q_i^{(2)}+q_i^{(3)})+q_i^{(4)}], p\leqslant 0\\ h\{q_i^{(1)}r_i^{(2)}+[-3(q_i^{(1)}+p_i\omega_i^{(1)})+2(q_i^{(2)}+p_i\omega_i^{(2)})\\ \quad+2(q_i^{(3)}+p_i\omega_i^{(3)})-(q_i^{(4)}+p_i\omega_i^{(4)})]r_i^{(3)}\\ \quad+4[(q_i^{(1)}+p_i\omega_i^{(1)})-(q_i^{(2)}+p_i\omega_i^{(2)})-(q_i^{(3)}+p_i\omega_i^{(3)})\\ \quad+(q_i^{(4)}+p_i\omega_i^{(4)})]r_i^{(4)}\}, p_i>0\end{cases}$$

式中

$$p_i=\frac{q_i^{(3)}-q_i^{(2)}}{\omega_i^{(3)}-\omega_i^{(2)}}, i=0,1,\cdots,n-1$$

$$r_i^{(1)}=e^{-p_ih}, r_i^{(2)}=\frac{r_i^{(1)}-1}{-p_ih}, r_i^{(3)}=\frac{r_i^{(2)}-1}{-p_ih},$$

$$r_i^{(4)}=\frac{r_i^{(3)}-\frac{1}{2}}{-p_ih}, i=0,1,\cdots,n-1$$

$$\omega_i^{(1)}=y_{ij},\qquad q_i^{(1)}=f_i(t_j,\omega_0^{(1)},\omega_1^{(1)},\cdots,\omega_{n-1}^{(1)})$$

$$\omega_i^{(2)}=\omega_i^{(1)}+\frac{h}{2}q_i^{(1)}\qquad q_i^{(2)}=f_i\left(t_j+\frac{h}{2},\omega_0^{(2)},\omega_1^{(2)},\cdots,\omega_{n-1}^{(2)}\right)$$

$$\omega_i^{(3)}=\omega_i^{(1)}+\frac{h}{2}q_i^{(2)}\qquad q_i^{(3)}=f_i\left(t_j+\frac{h}{2},\omega_0^{(3)},\omega_1^{(3)},\cdots,\omega_{n-1}^{(3)}\right)$$

$$\omega_i^{(4)}=\omega_i^{(1)}+hq_i\qquad q_i^{(4)}=f_i(t_j+h,\omega_0^{(4)},\omega_1^{(4)},\cdots,\omega_{n-1}^{(4)}), i=0,1,\cdots,n-1$$

q_i 由公式

$$q_i=\begin{cases}q_i^{(3)}, p_i\leqslant 0\\ 2(q_i^{(3)}-q_i^{(1)})r_i^{(3)}+(q_i^{(1)}-q_i^{(2)})r_i^{(2)}+q_i^{(2)}, p_i>0\end{cases}\qquad i=0,1,\cdots,n-1$$

确定。

由上述计算公式可知，当 $p_i\leqslant 0$ 时，本方法便退化为四阶龙格－库塔方法。

本方法适合于求解刚性问题，但对一般的一阶微分方程组同样适用。

3. 函数原型

```
//积分一步特雷纳法
template <class _Ty>
void ODE_Treanor(_Ty t, _Ty h, valarray<_Ty>& y);
```

4. 参数

t：_Ty 类型变量。积分的起始点。

h：_Ty 类型变量。积分一步的步长。

n：整型变量。一阶微分方程组中方程个数，也是未知函数的个数。

y：valarray<_Ty>类型引用，长度为 n。存放在起始点 t 处的 n 个未知函数值 $y_i(t)$。返回时存放积分点 $t+h$ 处的 n 个未知函数值 $y_i(t+h)(i=0,1,\cdots,n-1)$。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,$

…,$n-1$) 的函数 FunctionValueT(),由用户自己编写,其形式如下:

```
template <class _Ty>
void FunctionValueT(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();   //微分方程组中方程的个数,也是未知函数的个数
      ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
      ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y'_0 = -21y_0 + 19y_1 - 20y_2, y_0(0) = 1.0 \\ y'_1 = 19y_0 - 21y_1 + 20y_2, \quad y_1(0) = 0.0 \\ y'_2 = 40y_0 - 40y_1 - 40y_2, \quad y_2(0) = -1.0 \end{cases}$$

用积分一步的特雷纳方法计算当 $h = 0.001$ 时,各积分点

$$t_i = i \times h, i = 0,1,\cdots,10$$

处的未知函数值 $y_0(t_i), y_1(t_i), y_2(t_i)(i = 0,1,\cdots,10)$。

主函数程序及计算微分方程组右端函数值的函数示例程序如下:

```
//ODE_Treanor.cpp          积分一步特雷纳法

#include <iostream>                                    //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"              //求解常微分方程(组)头文件
using namespace std;                                   //名字空间

void main(void)
{
    double y[3] = {1.0, 0.0, -1.0}, t(0.0), h(0.001);
    valarray<double> vy(y, 3);

    cout.setf(ios::fixed);          //输出数据为定点法
    cout.precision(6);              //精度 6 位

    cout << "t = " << t << endl;

    for(int j=0; j<3; j++)
       cout << " y(" << j << ") = " << y[j] << "\t";
    cout << endl;

    for(int i=0; i<10; i++)
    {
```

```
        ODE_Treanor(t,h,vy);
        t = t + h;
        cout << "t = " << t << endl;
        for(j = 0; j<3; j ++ )
            cout << " y(" << j << ") = " << vy[j] << "\t";
        cout << endl;
      }
        cout << endl;
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueT(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
      int n = y.size();   //微分方程组中方程的个数,也是未知函数的个数

      d[0] = -21.0 * y[0] + 19.0 * y[1] - 20.0 * y[2];
      d[1] = 19.0 * y[0] - 21.0 * y[1] + 20.0 * y[2];
      d[2] = 40.0 * y[0] - 40.0 * y[1] - 40.0 * y[2];
}
```

8. 示例结果

```
t = 0.000000
    y(0) = 1.000000  y(1) = 0.000000    y(2) = -1.000000
t = 0.001000
    y(0) = 0.998243  y(1) = -0.000241  y(2) = -0.921600
t = 0.002000
    y(0) = 0.995011  y(1) = 0.000997    y(2) = -0.845389
t = 0.003000
    y(0) = 0.990406  y(1) = 0.003612    y(2) = -0.772922
t = 0.004000
    y(0) = 0.984554  y(1) = 0.007478    y(2) = -0.703802
t = 0.005000
    y(0) = 0.977570  y(1) = 0.012480    y(2) = -0.637904
t = 0.006000
    y(0) = 0.969562  y(1) = 0.018510    y(2) = -0.575151
t = 0.007000
    y(0) = 0.960629  y(1) = 0.025469    y(2) = -0.515481
t = 0.008000
    y(0) = 0.950869  y(1) = 0.033258    y(2) = -0.458830
t = 0.009000
    y(0) = 0.940374  y(1) = 0.041788    y(2) = -0.405136
t = 0.010000
    y(0) = 0.929229  y(1) = 0.050971    y(2) = -0.354333
```

本问题的解析解为

$$\begin{cases} y_0(t) = \dfrac{1}{2}e^{-2t} + \dfrac{1}{2}e^{-40t}(\cos 40t + \sin 40t) \\ y_1(t) = \dfrac{1}{2}e^{-2t} - \dfrac{1}{2}e^{-40t}(\cos 40t + \sin 40t) \\ y_2(t) = -e^{-40t}(\cos 40t - \sin 40t) \end{cases}$$

11.2.15 全区间积分特雷纳法

1. 功能

用特雷纳(Treanor)方法对一阶刚性(stiff)微分方程组进行全区间积分。

2. 算法

同本章 11.2.14 节算法。

3. 函数原型

```
//全区间积分特雷纳法
template <class _Ty>
void ODE_TreanorInterval(_Ty t, _Ty h, valarray<_Ty>& y,int k, matrix<_Ty>& z);
```

4. 参数

t:_Ty 类型变量。积分起始点。

h:_Ty 类型变量。积分一步的步长。

y:valarray<_Ty>类型引用,长度为 n。存放在起始点 t 处的 n 个未知函数值 $y_i(t)$(即初值)。返回时其值在二维数组 z 的第零列中。即

$$z(i,0) = y_i(t), \qquad i = 0,1,\cdots,n-1$$

k:整型变量。积分的步数(包括起始点这一步)。

z:matrix<_Ty>引用,体积为 $n \times k$。返回 k 个积分点处的 n 个未知函数值,即

$$z(i,j) = y_i(t_j), i = 0,1,\cdots,n-1; j = 0,1,\cdots,k-1$$

其中 $z(i,0) = y_i(t) i = 0,1,\cdots,n-1$ 为初值 。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i = 0,1,\cdots,n-1)$ 的函数 FunctionValueT(),由用户自己编写,其形式如下:

```
template <class _Ty>
void FunctionValueT(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();       //微分方程组中方程的个数,也是未知函数的个数
        ......
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
        ......
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(2) 输入的数据应为浮点数类型。

7. 示例

设一阶刚性方程组及初值为

$$\begin{cases} y'_0 = -21y_0 + 19y_1 - 20y_2, y_0(0) = 1.0 \\ y'_1 = 19y_0 - 21y_1 + 20y_2, \quad y_1(0) = 0.0 \\ y'_2 = 40y_0 - 40y_1 - 40y_2, \quad y_2(0) = -1.0 \end{cases}$$

用全区间积分的特雷纳方法计算当 $h=0.001$ 时，各积分点

$$t_i = i \times h, i = 0,1,\cdots,101$$

处的未知函数值 $y_0(t_i), y_1(t_i), y_2(t_i)(i = 0,1,\cdots,101)$。

主函数程序及计算微分方程组右端函数值的函数示例程序如下：

```
//ODE_TreanorInterval.cpp      全区间积分特雷纳法

#include <iostream>                                  //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"            //求解常微分方程(组)头文件
using namespace std;                                 //名字空间

void main(void)
{
    double y[3] = {1.0, 0.0, -1.0}, t(0.0), h(0.001);
    valarray<double> vy(y, 3);
    matrix<double> vz(3,101);

    cout.setf(ios::fixed);          //输出数据为定点法
    cout.precision(6);              //精度 6 位

    ODE_TreanorInterval(t,h,vy,101,vz);

    for(int i=0; i<101; i+=10)
    {
      t = h * i;
      cout << "t = " << t << endl;
      for(int j=0; j<3; j++)
          cout << " y(" << j << ") = " << vz(j, i) << "\t";
      cout << endl;
    }
    cout << endl;
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueTI(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
      int n = y.size();      //微分方程组中方程的个数，也是未知函数的个数
```

```
    d[0] = -21.0 * y[0] + 19.0 * y[1] - 20.0 * y[2];
    d[1] = 19.0 * y[0] - 21.0 * y[1] + 20.0 * y[2];
    d[2] = 40.0 * y[0] - 40.0 * y[1] - 40.0 * y[2];
}
```

8. 示例结果

```
t = 0.000000
    y(0) = 1.000000  y(1) = 0.000000  y(2) = -1.000000
t = 0.010000
    y(0) = 0.929229  y(1) = 0.050971  y(2) = -0.354333
t = 0.020000
    y(0) = 0.797831  y(1) = 0.162959  y(2) = 0.010860
t = 0.030000
    y(0) = 0.665430  y(1) = 0.276335  y(2) = 0.172648
t = 0.040000
    y(0) = 0.559130  y(1) = 0.363987  y(2) = 0.208260
t = 0.050000
    y(0) = 0.485243  y(1) = 0.419592  y(2) = 0.179452
t = 0.060000
    y(0) = 0.440267  y(1) = 0.446652  y(2) = 0.127956
t = 0.070000
    y(0) = 0.415985  y(1) = 0.453371  y(2) = 0.077326
t = 0.080000
    y(0) = 0.404428  y(1) = 0.447714  y(2) = 0.037909
t = 0.090000
    y(0) = 0.399326  y(1) = 0.435943  y(2) = 0.012040
t = 0.100000
    y(0) = 0.396487  y(1) = 0.422242  y(2) = -0.002139
```

本问题的解析解为

$$\begin{cases} y_0(t) = \dfrac{1}{2}e^{-2t} + \dfrac{1}{2}e^{-40t}(\cos 40t + \sin 40t) \\ y_1(t) = \dfrac{1}{2}e^{-2t} - \dfrac{1}{2}e^{-40t}(\cos 40t + \sin 40t) \\ y_2(t) = -e^{40t}(\cos 40t - \sin 40t) \end{cases}$$

11.2.16 积分刚性方程组吉尔法

1. 功能

用吉尔(Gear)方法积分一阶常微分方程组的初值问题。

2. 算法

设一阶微分方程组为

$$\begin{cases} y'_0 = f_0(t, y_0, y_1, \cdots, y_{n-1}), & y(t_0) = y_{00} \\ y'_1 = f_1(t, y_0, y_1, \cdots, y_{n-1}), & y_1(t_0) = y_{10} \\ \quad \vdots \\ y'_{n-1} = f_{n-1}(t, y_0, y_1, \cdots, y_{n-1}), & y_{n-1}(t_0) = y_{n-1,0} \end{cases}$$

吉尔方法的计算公式如下：

预报　$Z_{i,(0)} = PZ_{i-1}$

校正　$Z_{i,(j+1)} = Z_{i,(j)} - L\left[\left[l_1 I - l_0 \frac{\partial F}{\partial Y}\right]^{-1} G(Z_{i,(j)})\right]^{\mathrm{T}}$

终值　$Z_i = Z_{i,(M)}$

其中

$$Z_i = (Y_i, hY_i', \cdots, h^p Y_i^{(p)}/p!)^{\mathrm{T}}$$

为$(p+1)\times n$的矩阵；

$$G(Z_{i,(j)}) = hF(t_i, Y_{i,(j)}) - hY_{i,(j)}'$$

为n维列向；$\frac{\partial F}{\partial Y}$为$n\times n$的雅可比(Jacobi)矩阵，即

$$\frac{\partial F}{\partial Y} = \begin{bmatrix} \frac{\partial f_0}{\partial y_0} & \frac{\partial f_0}{\partial y_1} & \cdots & \frac{\partial f_0}{\partial y_{n-1}} \\ \frac{\partial f_1}{\partial y_0} & \frac{\partial f_1}{\partial y_1} & \cdots & \frac{\partial f_1}{\partial y_{n-1}} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_{n-1}}{\partial y_0} & \frac{\partial f_{n-1}}{\partial y_1} & \cdots & \frac{\partial f_{n-1}}{\partial y_{n-1}} \end{bmatrix}$$

P为$(p+1)$阶的巴斯卡尔(PASCAL)三角矩阵，即

$$P = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 & 1 \\ & 1 & 2 & 3 & \cdots & k-1 & k \\ & & 1 & 3 & \cdots & \vdots & \vdots \\ & & & 1 & \cdots & \vdots & \vdots \\ & & & & \ddots & k-1 & k \\ & & & & & 1 & k \\ & & & & & & 1 \end{bmatrix}$$

L为$(p+1)$维列向量，即

$$L = (l_0, l_1, \cdots, l_p)^{\mathrm{T}}$$

在本函数中，$M=3$(即迭代校正3次)。向量L在本函数中自带。

上述计算过程是自开始、自动变步长且自动变阶的。

在本函数中，为了满足精度要求，首先考虑减小步长，只有当达到最小步长还没有满足精度要求时，才考虑提高方法的阶数。并且在用某个阶数的方法连续进行几步的计算中均满足精度要求时，考虑降低方法的阶数。

本函数能比较有效地积分刚性问题，也能积分非刚性问题。

3. 函数原型

```
//积分刚性方程组吉尔法
template <class _Ty>
int ODE_Gear(_Ty a, _Ty b, _Ty hmin, _Ty hmax, _Ty h, _Ty eps,
        valarray<_Ty>& y0, int k, valarray<_Ty>& t, matrix<_Ty>& z);
```

4. 参数

a:_Ty类型变量。积分的起始点。

b:_Ty 类型变量。积分区间的终点。

hmin:_Ty 类型变量。积分过程中允许的最小步长。

hmax:_Ty 类型变量。积分过程中允许的最大步长。

h:_Ty 类型变量。积分的拟定步长,在积分过程中将自动放大或缩小。hmin<<h<hmax。

eps:_Ty 类型变量。误差检验常数。

y:valarray<_Ty>类型引用,长度为 n。存放积分起始点 a 处的 n 个未知函数初值 $y_i(a)$。返回时其值在二维数组 z 的第零列;其余为本函数的工作单元。

k:整型变量。拟定输出的积分点数。

t: valarray<_Ty>类型引用,长度为 k。返回 k 个输出点处的自变量值(包括起始点)。

z:matrix<_Ty>引用,体积为 $n\times k$。返回 k 个积分点处的 n 个未知函数值,即

$$z(i,j)=y_i(t_j),i=0,1,\cdots,n-1;j=0,1,\cdots,k-1$$

其中 $z(i,0)i=0,1,\cdots,n-1$ 为初值 $y_i(a)$。

5. 返回值

+1:表示全区间积分成功。

-1:表示步长已小于最小步长 hmin,但精度未达到,积分停止(已算出的前若干点输出有效)。

-2:表示阶数已大于 6,积分停止(已算出的前若干点输出有效)。

-3:表示对于 $h\geqslant$ hmin 校正迭代不收敛,积分停止(已算出的前若干点输出有效)。

-4:表示对所处理的问题,要求的精度太高,积分停止(已算出的前若干点输出有效)。

6. 函数说明

本函数要调用以下 3 个函数:

(1) 全选主元高斯-约当法求矩阵逆的函数 MatrixInversionGS(),参看第 5 章 5.5.1 节。

(2) 本函数要调用计算微分方程组中各方程右端函数值 $f_i(t,y_0,y_1,\cdots,y_{n-1})(i=0,1,\cdots,n-1)$ 的函数 FunctionValueG(),由用户自己编写,其形式如下:

```
template <class _Ty>
void FunctionValueG(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
    int n = y.size();      //微分方程组中方程的个数,也是未知函数的个数
      ……
    d[0] = f0(t,y0,y1,…,yn-1) 的表达式
    d[1] = f1(t,y0,y1,…,yn-1) 的表达式
      ……
    d[n-1] = fn-1(t,y0,y1,…,yn-1) 的表达式
}
```

(3) 计算雅可比矩阵的函数 JacobiMatrix(),其形式如下:

```
template <class _Ty>
void JacobiMatrix(_Ty t, valarray<_Ty>& y, matrix<_Ty>& p)
{
   int n = y.size();      //微分方程组中方程的个数,也是未知函数的个数
   p[i][j] = p(i,j) = ∂fi/∂yj (i,j = 0,1,…,n-1) 的表达式;
}
```

(4) 输入的数据应为浮点数类型。

7. 示例

设一阶微分方程组及初值为

$$\begin{cases} y'_0 = -21y_0 + 19y_1 - 20y_2, y_0(0) = 1.0 \\ y'_1 = 19y_0 - 21y_1 + 20y_2, \quad y_1(0) = 0.0 \\ y'_2 = 40y_0 - 40y_1 - 40y_2, \quad y_2(0) = -1.0 \end{cases}$$

这是一个刚性方程组,其解析解为

$$\begin{cases} y_0(t) = \frac{1}{2}e^{-2t} + \frac{1}{2}e^{-40t}(\cos 40t + \sin 40t) \\ y_1(t) = \frac{1}{2}e^{-2t} - \frac{1}{2}e^{-40t}(\cos 40t + \sin 40t) \\ y_2(t) = -e^{-40t}(\cos 40t - \sin 40t) \end{cases}$$

用吉尔方法分以下四种情况求在区间[0,1]中的数值解。其中 $a = 0.0, b = 1.0, n = 3$,取 $k = 30$。

(1) $h = 0.01, h\min = 0.0001, h\max = 0.1, eps = 0.001$;

(2) $h = 0.01, h\min = 0.0001, h\max = 0.1, eps = 0.0001$;

(3) $h = 0.01, h\min = 0.00001, h\max = 0.1, eps = 0.0001$;

(4) $h = 0.01, h\min = 0.00001, h\max = 0.1, eps = 0.00001$。

主函数程序及计算微分方程组右端函数值的函数程序,计算雅可比矩阵函数的示例程序如下:

```
//ODE_Gear.cpp          积分刚性方程组吉尔法

#include<iostream>                              //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"      //求解常微分方程(组)头文件
using namespace std;                           //名字空间


void main(void)
{
      double a(0.0), b(1.0), h(0.01), hmax(0.1);
      double hmin[4] = {0.0001,0.0001,0.00001,0.00001};
      double eps[4] = {0.001,0.0001,0.0001,0.00001};
      matrix<double> z(3,30);
      valarray<double> y(3), t(30);

      cout.setf(ios::fixed);                   //输出数据为定点法
      cout.precision(6);                       //精度6位

      for(int k=0; k<4; k++)
      {
        y[0] = 1.0;
        y[1] = 0.0;
```

```
            y[2] = -1.0;

            int iReValue = ODE_Gear(a,b,hmin[k],hmax,h,eps[k],y,30,t,z);
            cout << "h = " << h << endl;
            cout << "hmin = " << hmin[k] << endl;
            cout << "hmax = " << hmax << endl;
            cout << "eps = " << eps[k] << endl << endl;
            cout << "iReValue = " << iReValue << endl << endl;

            for(int i = 0; i<30; i ++ )
            {
                cout << "t = " << t[i] << endl;
                for(int j = 0; j<3; j ++ )
                    cout << " y(" << j << ") = " << z(j, i) << "\t";
                cout << endl;
            }
            cout << endl;
        }
}

//计算微分方程组中各方程右端函数值
template <class _Ty>
void FunctionValueG(_Ty t, valarray<_Ty>& y, valarray<_Ty>& d)
{
      int n = y.size();      //微分方程组中方程的个数,也是未知函数的个数

   d[0] = -21.0 * y[0] + 19.0 * y[1] - 20.0 * y[2];
   d[1] = 19.0 * y[0] - 21.0 * y[1] + 20.0 * y[2];
   d[2] = 40.0 * y[0] - 40.0 * y[1] - 40.0 * y[2];

}

//计算雅可比矩阵的函数
template <class _Ty>
void JacobiMatrix(_Ty t, valarray<_Ty>& y, matrix<_Ty>& p)
{
        int n = y.size();      //微分方程组中方程的个数,也是未知函数的个数

        p(0,0) = -21.0;
        p(0,1) = 19.0;
        p(0,2) = -20.0;
        p(1,0) = 19.0;
        p(1,1) = -21.0;
        p(1,2) = 20.0;
```

```
        p(2,0) = 40.0;
        p(2,1) = -40.0;
        p(2,2) = -40.0;
}
```

8. 示例结果

```
h = 0.010000
hmin = 0.000100
hmax = 0.100000
eps = 0.001000

iReValue = 1

t = 0.000000
    y(0) = 1.000000  y(1) = 0.000000  y(2) = -1.000000
t = 0.000690
    y(0) = 0.998591  y(1) = 0.000032  y(2) = -0.946338
t = 0.001380
    y(0) = 0.996502  y(1) = 0.000745  y(2) = -0.894192
t = 0.003234
    y(0) = 0.988675  y(1) = 0.004881  y(2) = -0.759460
t = 0.005088
    y(0) = 0.976732  y(1) = 0.013146  y(2) = -0.635490
t = 0.006942
    y(0) = 0.961290  y(1) = 0.024924  y(2) = -0.522061
t = 0.010099
    y(0) = 0.928377  y(1) = 0.051630  y(2) = -0.352411
t = 0.013256
    y(0) = 0.889586  y(1) = 0.084253  y(2) = -0.210762
t = 0.016413
    y(0) = 0.847441  y(1) = 0.120268  y(2) = -0.094846
t = 0.019569
    y(0) = 0.803844  y(1) = 0.157775  y(2) = -0.002185
t = 0.023568
    y(0) = 0.748863  y(1) = 0.205096  y(2) = 0.085830
t = 0.027567
    y(0) = 0.696222  y(1) = 0.250139  y(2) = 0.145945
t = 0.031565
    y(0) = 0.647292  y(1) = 0.291531  y(2) = 0.183266
t = 0.035564
    y(0) = 0.602954  y(1) = 0.328392  y(2) = 0.202440
t = 0.039562
    y(0) = 0.563735  y(1) = 0.360192  y(2) = 0.207577
t = 0.045876
    y(0) = 0.512593  y(1) = 0.399741  y(2) = 0.195664
```

```
t = 0.052190
    y(0) = 0.473985  y(1) = 0.426901  y(2) = 0.169112
t = 0.058503
    y(0) = 0.446209  y(1) = 0.443372  y(2) = 0.136401
t = 0.064817
    y(0) = 0.427137  y(1) = 0.451281  y(2) = 0.103242
t = 0.071131
    y(0) = 0.414653  y(1) = 0.452743  y(2) = 0.073082
t = 0.078731
    y(0) = 0.405714  y(1) = 0.448597  y(2) = 0.043241
t = 0.086331
    y(0) = 0.400860  y(1) = 0.440564  y(2) = 0.021185
t = 0.093931
    y(0) = 0.398047  y(1) = 0.430683  y(2) = 0.006316
t = 0.101532
    y(0) = 0.395908  y(1) = 0.420320  y(2) = -0.002651
t = 0.109132
    y(0) = 0.393628  y(1) = 0.410287  y(2) = -0.007197
t = 0.119089
    y(0) = 0.389743  y(1) = 0.398321  y(2) = -0.008720
t = 0.129046
    y(0) = 0.384662  y(1) = 0.387864  y(2) = -0.007580
t = 0.139003
    y(0) = 0.378578  y(1) = 0.378716  y(2) = -0.005523
t = 0.148960
    y(0) = 0.371820  y(1) = 0.370543  y(2) = -0.003494
t = 0.158917
    y(0) = 0.364699  y(1) = 0.363027  y(2) = -0.001881

h = 0.010000
hmin = 0.000100
hmax = 0.100000
eps = 0.000100

iReValue = 1

t = 0.000000
    y(0) = 1.000000  y(1) = 0.000000  y(2) = -1.000000
t = 0.000218
    y(0) = 0.999707  y(1) = -0.000143 y(2) = -0.982702
t = 0.000436
    y(0) = 0.999341  y(1) = -0.000213 y(2) = -0.965556
t = 0.001286
    y(0) = 0.997381  y(1) = 0.000050  y(2) = -0.899883
```

```
t = 0.002137
    y(0) = 0.994389  y(1) = 0.001347  y(2) = -0.836513
t = 0.002987
    y(0) = 0.990432  y(1) = 0.003612  y(2) = -0.775437
t = 0.004397
    y(0) = 0.981900  y(1) = 0.009346  y(2) = -0.679183
t = 0.005806
    y(0) = 0.971188  y(1) = 0.017266  y(2) = -0.589113
t = 0.007216
    y(0) = 0.958607  y(1) = 0.027065  y(2) = -0.505108
t = 0.008626
    y(0) = 0.944427  y(1) = 0.038470  y(2) = -0.427027
t = 0.010509
    y(0) = 0.923405  y(1) = 0.055796  y(2) = -0.331659
t = 0.012393
    y(0) = 0.900503  y(1) = 0.075016  y(2) = -0.246140
t = 0.014276
    y(0) = 0.876207  y(1) = 0.095644  y(2) = -0.169995
t = 0.016160
    y(0) = 0.850950  y(1) = 0.117247  y(2) = -0.102715
t = 0.019069
    y(0) = 0.810926  y(1) = 0.151654  y(2) = -0.014984
t = 0.021978
    y(0) = 0.770725  y(1) = 0.186271  y(2) = 0.054870
t = 0.024887
    y(0) = 0.731274  y(1) = 0.220170  y(2) = 0.108895
t = 0.027797
    y(0) = 0.693289  y(1) = 0.252635  y(2) = 0.149087
t = 0.030706
    y(0) = 0.657317  y(1) = 0.283120  y(2) = 0.177348
t = 0.034273
    y(0) = 0.616518  y(1) = 0.317232  y(2) = 0.198334
t = 0.037840
    y(0) = 0.579765  y(1) = 0.347348  y(2) = 0.207108
t = 0.041407
    y(0) = 0.547237  y(1) = 0.373285  y(2) = 0.206393
t = 0.044975
    y(0) = 0.518920  y(1) = 0.395058  y(2) = 0.198575
t = 0.048542
    y(0) = 0.494658  y(1) = 0.412823  y(2) = 0.185694
t = 0.052901
    y(0) = 0.470134  y(1) = 0.429469  y(2) = 0.165550
t = 0.057260
    y(0) = 0.450670  y(1) = 0.441124  y(2) = 0.142891
```

```
t = 0.061619
     y(0) = 0.435577  y(1) = 0.448476  y(2) = 0.119615
t = 0.065979
     y(0) = 0.424148  y(1) = 0.452231  y(2) = 0.097089
t = 0.070338
     y(0) = 0.415701  y(1) = 0.453070  y(2) = 0.076241
t = 0.075490
     y(0) = 0.408704  y(1) = 0.451161  y(2) = 0.054520

h = 0.010000
hmin = 0.000010
hmax = 0.100000
eps = 0.000100

iReValue = 1

t = 0.000000
     y(0) = 1.000000  y(1) = 0.000000  y(2) = -1.000000
t = 0.000218
     y(0) = 0.999707  y(1) = -0.000143 y(2) = -0.982702
t = 0.000436
     y(0) = 0.999341  y(1) = -0.000213 y(2) = -0.965556
t = 0.001286
     y(0) = 0.997381  y(1) = 0.000050  y(2) = -0.899883
t = 0.002137
     y(0) = 0.994389  y(1) = 0.001347  y(2) = -0.836513
t = 0.002987
     y(0) = 0.990432  y(1) = 0.003612  y(2) = -0.775437
t = 0.004397
     y(0) = 0.981900  y(1) = 0.009346  y(2) = -0.679183
t = 0.005806
     y(0) = 0.971188  y(1) = 0.017266  y(2) = -0.589113
t = 0.007216
     y(0) = 0.958607  y(1) = 0.027065  y(2) = -0.505108
t = 0.008626
     y(0) = 0.944427  y(1) = 0.038470  y(2) = -0.427027
t = 0.010509
     y(0) = 0.923405  y(1) = 0.055796  y(2) = -0.331659
t = 0.012393
     y(0) = 0.900503  y(1) = 0.075016  y(2) = -0.246140
t = 0.014276
     y(0) = 0.876207  y(1) = 0.095644  y(2) = -0.169995
t = 0.016160
     y(0) = 0.850950  y(1) = 0.117247  y(2) = -0.102715
```

```
t = 0.019069
     y(0) = 0.810926  y(1) = 0.151654  y(2) = -0.014984
t = 0.021978
     y(0) = 0.770725  y(1) = 0.186271  y(2) = 0.054870
t = 0.024887
     y(0) = 0.731274  y(1) = 0.220170  y(2) = 0.108895
t = 0.027797
     y(0) = 0.693289  y(1) = 0.252635  y(2) = 0.149087
t = 0.030706
     y(0) = 0.657317  y(1) = 0.283120  y(2) = 0.177348
t = 0.034273
     y(0) = 0.616518  y(1) = 0.317232  y(2) = 0.198334
t = 0.037840
     y(0) = 0.579765  y(1) = 0.347348  y(2) = 0.207108
t = 0.041407
     y(0) = 0.547237  y(1) = 0.373285  y(2) = 0.206393
t = 0.044975
     y(0) = 0.518920  y(1) = 0.395058  y(2) = 0.198575
t = 0.048542
     y(0) = 0.494658  y(1) = 0.412823  y(2) = 0.185694
t = 0.052901
     y(0) = 0.470134  y(1) = 0.429469  y(2) = 0.165550
t = 0.057260
     y(0) = 0.450670  y(1) = 0.441124  y(2) = 0.142891
t = 0.061619
     y(0) = 0.435577  y(1) = 0.448476  y(2) = 0.119615
t = 0.065979
     y(0) = 0.424148  y(1) = 0.452231  y(2) = 0.097089
t = 0.070338
     y(0) = 0.415701  y(1) = 0.453070  y(2) = 0.076241
t = 0.075490
     y(0) = 0.408704  y(1) = 0.451161  y(2) = 0.054520

h = 0.010000
hmin = 0.000010
hmax = 0.100000
eps = 0.000010

iReValue = 1

t = 0.000000
    y(0) = 1.000000  y(1) = 0.000000  y(2) = -1.000000
t = 0.000069
     y(0) = 0.999923  y(1) = -0.000061  y(2) = -0.994497
```

```
t = 0.000138
    y(0) = 0.999839  y(1) = -0.000115  y(2) = -0.989009
t = 0.000531
    y(0) = 0.999241  y(1) = -0.000302  y(2) = -0.957984
t = 0.000924
    y(0) = 0.998407  y(1) = -0.000253  y(2) = -0.927454
t = 0.001317
    y(0) = 0.997344  y(1) = 0.000026  y(2) = -0.897417
t = 0.001956
    y(0) = 0.995144  y(1) = 0.000952  y(2) = -0.849651
t = 0.002595
    y(0) = 0.992388  y(1) = 0.002436  y(2) = -0.803183
t = 0.003234
    y(0) = 0.989108  y(1) = 0.004446  y(2) = -0.758009
t = 0.003872
    y(0) = 0.985335  y(1) = 0.006950  y(2) = -0.714120
t = 0.004810
    y(0) = 0.978964  y(1) = 0.011463  y(2) = -0.652041
t = 0.005747
    y(0) = 0.971679  y(1) = 0.016893  y(2) = -0.592682
t = 0.006684
    y(0) = 0.963562  y(1) = 0.023159  y(2) = -0.536005
t = 0.007621
    y(0) = 0.954694  y(1) = 0.030179  y(2) = -0.481972
t = 0.009414
    y(0) = 0.935907  y(1) = 0.045442  y(2) = -0.385808
t = 0.011207
    y(0) = 0.915141  y(1) = 0.062695  y(2) = -0.298804
t = 0.012999
    y(0) = 0.892843  y(1) = 0.081493  y(2) = -0.220566
t = 0.014792
    y(0) = 0.869413  y(1) = 0.101436  y(2) = -0.150669
t = 0.016585
    y(0) = 0.845210  y(1) = 0.122164  y(2) = -0.088667
t = 0.018378
    y(0) = 0.820554  y(1) = 0.143358  y(2) = -0.034099
t = 0.020171
    y(0) = 0.795725  y(1) = 0.164737  y(2) = 0.013504
t = 0.021963
    y(0) = 0.770970  y(1) = 0.186055  y(2) = 0.054617
t = 0.023756
    y(0) = 0.746499  y(1) = 0.207100  y(2) = 0.089715
t = 0.025549
    y(0) = 0.722493  y(1) = 0.227693  y(2) = 0.119264
```

```
t = 0.027342
    y(0) = 0.699104  y(1) = 0.247681  y(2) = 0.143724
t = 0.029134
    y(0) = 0.676459  y(1) = 0.266938  y(2) = 0.163540
t = 0.030927
    y(0) = 0.654658  y(1) = 0.285362  y(2) = 0.179142
t = 0.032720
    y(0) = 0.633784  y(1) = 0.302872  y(2) = 0.190946
t = 0.034513
    y(0) = 0.613895  y(1) = 0.319408  y(2) = 0.199344
t = 0.037424
    y(0) = 0.583809  y(1) = 0.344076  y(2) = 0.206683
```

11.2.17 二阶微分方程边值问题数值解法

1. 功能

用有限差分法求二阶线性微分方程边值问题的数值解。

2. 算法

设线性二阶微分方程的边值问题为

$$\begin{cases} u(x)y'' + v(x)y' + \omega(x)y = f(x) \\ y(a) = \alpha \\ y(b) = \beta \end{cases}$$

现要求未知函数 $y(x)$ 在区间$[a,b]$上的n个(包括端点 $x=a$ 与 $x=b$) 等距离散点上的近似解。其中 n 个等距散点为：

$$x_i = a + i \cdot h \qquad i = 0,1,\cdots,n-1$$
$$h = (b-a)/(n-1)$$

显然，$y(x_0) = y(a) = \alpha, y(x_{n-1}) = y(b) = \beta$。

用中心差分近似代替 $y''(x_i)$ 与 $y'(x_i)$，即

$$y'(x_i) \approx \frac{y_{i+1} - y_{i-1}}{2h}$$

$$y'(x_i) \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}$$

其中 $y_i = y(x_i)$。将它们代入原微分方程得

$$\frac{u(x_i)}{h^2}(y_{i+1} - 2y_i + y_{i-1}) + \frac{v(x_i)}{2h}(y_{i+1} - y_{i-1}) = \omega(x_i)y_i = f(x_i)$$

$$i = 1,2,\cdots,n-2$$

整理后得到如下关于 $y_i(i = 0,1,\cdots,n-1)$ 的方程组

$$\begin{cases} y_0 = \alpha \\ p_i y_{i-1} + q_i y_i + r_i y_{i+1} = d_i, \qquad i = 1,2,\cdots,n-2 \\ y_{n-1} = \beta \end{cases}$$

其中

$$\left.\begin{aligned} p_i &= u(x_i) - \frac{h}{2}v(x_i) \\ q_i &= h^2\omega(x_i) - 2u(x_i) \\ r_i &= u(x_i) + \frac{h}{2}v(x_i) \\ d_i &= h^2 f(x_i) \end{aligned}\right\} \quad i = 1,2,\cdots,n-2$$

用矩阵表示为

$$\begin{bmatrix} q_0 & r_0 & & 0 & \\ p_1 & q_1 & r_1 & & \\ & \ddots & \ddots & \ddots & \\ & & p_{n-2} & q_{n-2} & r_{n-2} \\ 0 & & & p_{n-1} & q_{n-1} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-2} \\ d_{n-1} \end{bmatrix}$$

其中 $q_0 = 1, r_0 = 0, d_0 = \alpha; p_{n-1} = 0, q_{n-1} = 1, d_{n-1} = \beta$。

此为三对角线方程组，可用追赶法求解。

在实际计算时，为了提高精度，用步长 $h = (b-a)/(n-1)$ 计算得到 $n-2$ 个等距离散点上的一组近似解：

$$y_i^{(h)} = y_i[a + i \cdot h], \qquad i = 1,2,\cdots,n-2$$

再用步长 $\frac{h}{2} = (b-a)/(2n-2)$ 得到 $n-2$ 个等距离散点上的另一组近似解：

$$y_i^{(\frac{h}{2})} = y_i[a + i \cdot h], \qquad i = 1,2,\cdots,n-2$$

最后令

$$y_i = \frac{4y_i^{(h/2)} - y_i^{(h)}}{3}, \qquad i = 1,2,\cdots,n-2$$

3. 函数原型

```
//二阶微分方程边值问题数值解法
template <class _Ty>
void ODE_LinearBoundaryValude(_Ty a, _Ty b, _Ty ya, _Ty yb, int n, valarray<Ty>& y);
```

4. 参数

a：_Ty 类型变量。求积分区间的左端点。

b：_Ty 类型变量。求积分区间的右端点。

ya：_Ty 类型变量。未知函数在求解区间左端点处的函数值 $y(a)$。

yb：_Ty 类型变量。未知函数在求解区间右端点处的函数值 $y(b)$。

n：整型变量。求区间$[a,b]$的等分点数(包括左、右端点 a,b)，即各等分点为

$$x_i = a + i \cdot h \qquad i = 0,1,\cdots,n-1$$

其中 $h = (b-a)/(n-1)$。

y：valarray<_Ty>类型引用，长度为 n。返回 n 个等距离散点上的未知函数值 $y(x_i)(i = 0,1,\cdots,n-1)$。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数调用追赶求解三对角线方程组的函数 LE_TridiagonalEquationGauss()，参看

第 7 章 7.2.3 节。

(2) 本函数还要调用计算二阶微分方程中的函数 $u(x),v(x),\omega(x)$ 和 $f(x)$ 的值的函数 FunctionValueLBV(),由用户自己编写,其形式如下:

```
template <class _Ty>
void FunctionValueLBV(_Ty x, valarray<_Ty>& z)
{
    z[0]= u(x) 的表达式;
    z[1]= v(x) 的表达式;
    z[2]= w(x) 的表达式;
    z[3]= f(x) 的表达式;
}
```

(3) 输入的数据应为浮点数类型。

7. 示例

设二阶微分方程边值问题为

$$\begin{cases} -y''+\dfrac{2}{x^2}y=\dfrac{1}{x} \\ y(2)=0,y(3)=0 \end{cases}$$

求区间[2,3]等分数为 $n=11$(步长 $h=0.1$)。其中 $u(x)=-1,v(x)=0,\omega(x)=\dfrac{2}{x^2},f(x)=\dfrac{1}{x}$。

主函数程序及计算 $u(x),v(x),\omega(x)$ 和 $f(x)$ 值的函数示例程序如下:

```
//ODE_LinearBoundaryValude.cpp      定步长全区间积分的欧拉法

#include <iostream>                                    //输入输出流头文件
#include "OrdinaryDifferentialEguation.h"              //求解常微分方程(组)头文件
using namespace std;                                   //名字空间

void main(void)
{
      double a(2.0), b(3.0), ya(0.0), yb(0.0);
      valarray<double> y(11);

      cout.setf(ios::fixed);                           //输出数据为定点法
      cout.precision(6);                               //精度 6 位

      ODE_LinearBoundaryValude(a,b,ya,yb,11,y);

      for(int i=0; i<11; i++)
        cout << " y(" << i << ") = " << y[i] << endl;
      cout << endl;
}

//计算二阶微分方程组函数
template <class _Ty>
```

```
void FunctionValueLBV(_Ty x, valarray<_Ty>& z)
{
    z[0] = -1.0;
    z[1] = 0.0;
    z[2] = 2.0/(x*x);
    z[3] = 1.0/x;
}
```

8. 示例结果

```
y(0) = 0.000000
y(1) = 0.018609
y(2) = 0.032536
y(3) = 0.042048
y(4) = 0.047368
y(5) = 0.048684
y(6) = 0.046154
y(7) = 0.039912
y(8) = 0.030075
y(9) = 0.016742
y(10) = 0.000000
```

本问题的解析解为

$$y(x) = \left(19x - 5x^2 - \frac{36}{x}\right)/38$$

第 12 章　数值拟合与逼近

本章介绍有关数值拟合与逼近的算法。每种算法都用C++语言以函数的形式实现。12.1 节介绍数值拟合与逼近的基本概念，以及数值拟合与逼近算法程序原型的头文件，12.2 节给出了数值拟合与逼近的使用示例及相关说明。

12.1　基本概念与头文件

本节介绍数值拟合与逼近的基本概念，以及各种数值拟合与逼近算法的函数程序原型，这些原型包含在头文件 FittingApproximation.h 中，函数的定义内容在 FittingApproximation.inl 文件中。

本章介绍的数值拟合与逼近算法及程序包括：

- 最小二乘曲线拟合；
- 切比雪夫曲线拟合；
- 最佳一致逼近多项式里米兹法；
- 矩形域的最小二乘曲面拟合。

12.1.1　数值拟合与逼近的基本概念

1. 拟合与逼近

在科学研究与工程计算中得到的原始数据往往只是在某些点上的离散值，它们所代表的函数关系不易找到一个容易表示的数学表达式；或者所得到的数据的函数表达式比较复杂，不易计算，这样在计算这些函数其他所需要的数值方面带来了种种不便。解决这个难题的方法之一就是利用一些性质相对“好”的简单函数，在某种规定和标准之下，去拟合或逼近这些“困难”函数，然后通过这些简单函数去获得所希望得到的结果。

2. 拟合与逼近函数

拟合与逼近函数就是前面提到的“好”的简单函数。常用的拟合与逼近函数有多项式和三角函数。使用多项式作为拟合或逼近函数，则称该多项式为“拟合多项式”或“逼近多项式”。

本章仅介绍利用多项式进行拟合与逼近。有关多项式的介绍请参阅第 4 章“多项式与连分式函数计算”。

12.1.2　FittingApproximation.h

```
//FittingApproximation.h  拟合与逼近头文件
// Ver 1.0.0.0
// 版权所有(C) 何渝，2002
// 最后修改：2002.9.6
```

```
#ifndef _FITTINGAPPROXIMATION_H        //避免多次编译
#define _FITTINGAPPROXIMATION_H

#include <valarray>          //模板类数组的标准头文件
#include <Matrix.h>          //模板类矩阵头文件
#include <comm.h>            //公共头文件
using namespace std;         //名字空间

//最小二乘曲线拟合
template <class _Ty>
void FitCurveLeastSquares(valarray<_Ty>& x, valarray<_Ty>& y,
                          valarray<_Ty>& a, valarray<_Ty>& dt);

//切比雪夫曲线拟合
template <class _Ty>
void FitCurveChebyshev(valarray<_Ty>& x, valarray<_Ty>& y, valarray<_Ty>& a);

//最佳一致逼近多项式里米兹法
template <class _Ty>
void ApproximationRemez(_Ty a, _Ty b, valarray<_Ty>& p,_Ty eps);

//矩形域的最小二乘曲面拟合
template <class _Ty>
void FitSurfaceLeastSquares(valarray<_Ty>& x, valarray<_Ty>& y,
            matrix<_Ty>& z, matrix<_Ty>& a,valarray<_Ty>& dt);

#include "FittingApproximation.inl"      //类及相关函数的定义头文件

#endif      // _FITTINGAPPROXIMATION_H
```

12.2 数值拟合与逼近计算方法

12.2.1 最小二乘曲线拟合

1. 功能

用最小二乘法求给定数据点的拟合多项式。

2. 算法

设已知 n 个数据点$(x_i,y_i)(i=0,1,\cdots,n-1)$,求$(m-1)$次最小二乘拟合多项式

$$P_{m-1}(x)=a_0+a_1x+a_2x^2+\cdots+a_{m-1}x^{m-1}$$

其中 $m\leqslant n$ 且 $m\leqslant 20$。

设拟合多项式为各正交多项式 $Q_j(x)(j=0,1,\cdots,m-1)$的线性组合:

$$P_{m-1}(x)=c_0Q_0(x)+c_1Q_1(x)+\cdots+c_{m-1}Q_{m-1}(x)$$

其中 $Q_j(x)$可以由以下递推公式来构造：

$Q_0(x)=1$

$Q_1(x)=(x-a_1)$

$Q_{j+1}(x)=(x-a_{j+1})Q_j(x)-\beta_j Q_{j-1}(x),\quad j=1,2,\cdots,m-2$

若设

$$d_j=\sum_{i=0}^{n-1}Q_j^2(x_i),\quad j=0,1,\cdots,m-1$$

则

$$\left.\begin{aligned}\alpha_{j+1}&=\frac{1}{d_j}\sum_{i=0}^{n-1}x_iQ_j^2(x_i)\\ \beta_j&=d_j/d_{j-1}\end{aligned}\right\},\quad j=0,1,\cdots,m-2$$

可以证明，由上述递推构造的多项式函数组$\{Q_j(x)\}(j=0,1,\cdots,m-1)$是互相正交的。根据最小二乘原理，可得

$$c_j=\frac{1}{d_j}\sum_{i=0}^{n-1}y_iQ_j(x_i),\quad j=0,1,\cdots,m-1$$

最后可以化成一般的 $m-1$ 次多项式

$$P_{m-1}(x)=a_0+a_1x+a_2x^2+\cdots+a_{m-1}x^{m-1}$$

由于拟合多项式的次数越高，其拟合精度未必越高，因此，在本函数中允许的拟合次数最高为 19(即 m 的最大值为 20)。

具体计算步骤如下。

(1) $Q_0(x)=1$，可得

$$b_0=1,d_0=n,c_0=\frac{1}{d_0}\sum_{i=0}^{n-1}y_i,\quad \alpha=\frac{1}{d_1}\sum_{i=0}^{n-1}x_i$$

$$a_0=c_0b_0$$

(2) $Q_1(x)=(x-\alpha)$，可得

$$t_0=-\alpha,\quad t_1=1$$

$$d_1=\sum_{i=0}^{n-1}Q_1^2(x_i),c_1=\frac{1}{d_1}\sum_{i=0}^{n-1}y_iQ_1(x_i)$$

$$\alpha=\frac{1}{d_1}\sum_{i=0}^{n-1}x_iQ_1^2(x_i),\beta=d_1/d_0$$

$$a_0=a_0+c_1t_0,a_1=c_1t_1$$

(3) 对于 $j=2,3,\cdots,m-1$ 作以下各步

$$\begin{aligned}Q_j(x)&=(x-\alpha)Q_{j-1}(x)-\beta Q_{j-2}(x)\\&=(x-\alpha)(t_{j-1}x^{j-1}+\cdots+t_1x+t_0)-\beta(b_{j-2}x^{j-2}+\cdots+b_1x+b_0)\\&=s_jx^j+s_{j-1}x^{j-1}+\cdots+s_1x+s_0\end{aligned}$$

其中 $s_i(i=0,1,\cdots,j)$由以下递推公式计算：

$$\begin{cases}s_j=t_{j-1}\\ s_{j-1}=-\alpha t_{j-1}+t_{j-2}\\ s_k=-\alpha t_k+t_{k-1}-\beta b_k,k=j-2,\cdots,1\\ s_0=-\alpha t_0-\beta t_0\end{cases}$$

再计算

$$d_j = \sum_{i=0}^{n-1} Q_j^2(x_i)$$

$$c_j = \sum_{i=0}^{n-1} y_i Q_j(x_i)/d_j$$

$$\alpha = \sum_{i=0}^{n-1} x_i Q_j^2(x_i)/d_j$$

$$\beta = d_j/d_{j-1}$$

由此可以计算相应的 a_i：

$$\begin{cases} a_j = c_j s_j \\ a_k = a_k + c_j s_k, k = j-1, \cdots, 1, 0 \end{cases}$$

且

$$\begin{cases} t_j = s_j \\ b_k = t_k, t_k = s_k, k = j-1, \cdots, 1, 0 \end{cases}$$

在实际计算过程中，为了防止运算溢出，x_i 用

$$x_i^* = x_i - \bar{x}, i = 0, 1, \cdots, n-1$$

代替，其中

$$\bar{x} = \sum_{i=0}^{n-1} x_i/n$$

此时，拟合多项式的形式为

$$P_{m-1}(x) = a_0 + a_1(x-\bar{x}) + a_2(x-\bar{x})^2 + \cdots + a_{m-1}(x-\bar{x})^{m-1}$$

3. 函数原型

```
//最小二乘曲线拟合
template <class _Ty>
void FitCurveLeastSquares(valarray<_Ty>& x, valarray<_Ty>& y,
                          valarray<_Ty>& a, valarray<_Ty>& dt);
```

4. 参数

x：valarray<_Ty>类型引用，长度为 n，存放给定 n 个数据点的 x 坐标。

y：valarray<_Ty>类型引用，长度为 n，存放给定 n 个数据点的 y 坐标。

a：valarray<_Ty>类型引用，长度为 m，返回 $m-1$ 次拟合多项式的 m 个系数。

dt：valarray<_Ty>类型引用，长度为 3。其中，$dt(0)$返回拟合多项式与数据点误差的平方和；$dt(1)$返回拟合多项式与数据误差的绝对值之和；$dt(2)$返回拟合多项式与数据点误差绝对值的最大值。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 设给定的数据点为 n 个数。

(2) 拟合多项式的项数设为 m，即拟合多项式的最高次数为 $m-1$。要求 $m \leqslant n$ 且 $m \leqslant 20$。若 $m > n$ 或 $m > 20$，则本函数自动按 $m = \min\{n, 20\}$处理。

(3) 输入、输出的数据应为浮点数类型。

7. 示例

设给定函数

$$f(x) = x - e^{-x}$$

从 $x_0 = 0$ 开始，取步长 $h = 0.1$ 的 20 个数据点，求 5 次最小二乘拟合多项式

$$P_5(x) = a_0 + a_1(x - \bar{x}) + a_2(x - \bar{x})^2 + \cdots + a_5(x - \bar{x})^5$$

其中

$$\bar{x} = \sum_{i=0}^{19} x_i / 20 = 0.95$$

主函数示例程序如下：

```
//FitCurveLeastSquares.cpp        最小二乘曲线拟合

#include <iostream>                  //模板类输入输出流标准头文件
#include "FittingApproximation.h"    //拟合与逼近头文件
using namespace std;                 //名字空间

void main(void)
{
  valarray<double> x(20), y(20), da(6), dt(3);

  for (int i = 0; i<20; i++)
  {
     x[i] = 0.1 * i;
      y[i] = x[i] - exp(-x[i]);
  }

  cout.setf(ios::fixed);   //输出数据为定点法
  cout.precision(14);      //精度 14 位

  FitCurveLeastSquares(x,y,da,dt);

  for (i = 0; i<6; i++)
     cout << "da[" << i << "] = " << da[i] << endl;

  cout << endl;

  for (i = 0; i<3; i++)
     cout << "dt[" << i << "] = " << dt[i] << endl;
}
```

8. 示例结果

```
da[0] = 0.56324804925759
da[1] = 1.38674670124567
da[2] = -0.19313388870525
da[3] = 0.06440354946793
```

```
da[4] = -0.01684122038873
da[5] = 0.00334428831003

dt[0] = 0.00000000180174
dt[1] = 0.00016850492746
dt[2] = 0.00001539396205
```

12.2.2 切比雪夫曲线拟合

1. 功能

给定 n 个数据点，求切比雪夫(Chebyshev)意义下的最佳拟合多项式。

2. 算法

设已知 n 个数据点

$$(x_i, y_i), i = 0, 1, \cdots, n-1$$

其中 $x_0 < x_1 < \cdots < x_{n-1}$。求 $m-1$ 次($m<n$ 或 $n \leqslant 20$)多项式

$$P_{m-1}(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{m-1} x^{m-1}$$

使得在 n 个给定点上的偏差最大值为最小，即

$$\max_{0 \leqslant i \leqslant n-1} | P_{m-1}(x_i) - y_i | = \min$$

其计算步骤如下。

从给定的 n 个点中选取 $m+1$ 个不同点 $u_0, u_1, \cdots, u_m$ 组成初始参考点集。

设在初始点集 $u_0, u_1, \cdots, u_m$ 上，参考多项式 $\Phi(x)$ 的偏差为 h，即参考多项式 $\Phi(x)$ 在初始点集上的取值为

$$\Phi(u_i) = f(u_i) + (-1)^i h, i = 0, 1, \cdots, m$$

且 $\Phi(u_i)$ 的各阶差商是 h 的线性函数。

由于 $\Phi(x)$ 为 $m-1$ 次多项式，其 m 阶差商等于零，由此可以求出 h。再根据 $\Phi(u_i)$ 的各阶差商，由牛顿插值公式可求出 $\Phi(x)$

$$\Phi(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{m-1} x^{m-1}$$

令

$$hh = \max_{0 \leqslant i \leqslant n-1} | \Phi(x_i) - y_i |$$

若 $hh = h$，则 $\Phi(x)$ 即为所求的拟合多项式。

若 $hh > h$，则用达到偏差最大值的点 x_i 代替点集 $\{u_i\}(i=0,1,\cdots,m)$ 中离 x_i 最近且具有与

$$\Phi(x_i) - y_i$$

的符号相同的点，从而构造一个新的参考点集。用这个新的参考点集重复以上过程，直到最大逼近误差等于参考偏差为止。

3. 函数原型

```
//切比雪夫曲线拟合
template <class _Ty>
void FitCurveChebyshev(valarray<_Ty>& x, valarray<_Ty>& y, valarray<_Ty>& a);
```

4. 参数

x：valarray<_Ty>类型引用，长度为 n，存放 n 个数据点的 x 坐标。

y：valarray<_Ty>类型引用，长度为 n，存放 n 个数据点的 y 坐标。

a：双精度一维数组，长度为 $m+1$，其中 m 元素（$a(0)$到 $a(m-1)$）返回 $m-1$ 次拟合多项式

$$P_{m-1} = a_0 + a_1 x + a_2 x^2 + \cdots + a_{m-1} x^{m-1}$$

的系数 $a_i(i=0,1,\cdots,m-1)$；最后一个元素 $a(m)$返回拟合多项式 $P_{m-1}(x)$的偏差最大值。若 $a(m)$为负值，说明在迭代过程中参考偏差不再增大，其绝对值为当前选择的参考偏差。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 设给定的数据点为 n 个数。

(2) 拟合多项式的项数设为 m，即拟合多项式的最高次数为 $m-1$。要求 $m<n$ 且 $m\leqslant 20$。若 $m\geqslant n$ 或 $m>20$，则本函数自动按 $m=\min\{n-1,20\}$处理。

(3) 输入、输出的数据应为浮点数类型。

7. 示例

取 $f(x)=\text{arctg}x$ 在区间[−1,1]上的 101 个点

$$x_i = -1.0 + i \cdot 0.02, \qquad i = 0,1,\cdots,100$$

上的函数值 $y_i=f(x_i)(i=0,1,\cdots,100)$。根据此 101 个数据点构造切比雪夫意义下的五次拟合多项式

$$P_5(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5$$

主函数示例程序如下：

```
//FitCurveChebyshev.cpp        切比雪夫曲线拟合

#include <iostream>                  //模板类输入输出流标准头文件
#include "FittingApproximation.h" //拟合与逼近头文件
using namespace std;                 //名字空间

void main(void)
{
  valarray<double> x(101), y(101), da(7);

  for(int i=0; i<101; i++)
  {
     x[i] = -1.0 + 0.02 * i;
      y[i] = atan(x[i]);
   }

  FitCurveChebyshev(x,y,da);

  for (i=0; i<6; i++)
     cout << "da[" << i << "] = " << da[i] << endl;

  cout << endl << "MAX(p-f) = " << da[6] << endl;
}
```

8. 示例结果

```
da[0] = -1.11022e-016
da[1] = 0.995364
da[2] = 0
da[3] = -0.288716
da[4] = -2.77556e-017
da[5] = 0.0793575

MAX(p-f) = 0.000607072
```

12.2.3 最佳一致逼近多项式里米兹法

1. 功能

用里米兹(Remez)方法求给定函数的最佳逼近多项式。

2. 算法

若函数 $f(x)$在区间$[a,b]$上的 $n-1$ 次最佳一致逼近多项式为

$$P_{n-1}(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{n-1} x^{n-1}$$

则存在 $n+1$ 个点的交错点组 $|x_i|$ 满足

$$f(x_i) - P_{n-1}(x_i) = (-1)^i \mu, i = 0,1,\cdots,n$$

或

$$f(x_i) - P_{n-1}(x_i) = (-1)^{i+1} \mu, i = 0,1,\cdots,n$$

其中

$$\mu = \max_{x \in [a,b]} |f(x) - P_{n-1}(x)| 。$$

求函数 $f(x)$在区间$[a,b]$上的 $n-1$ 次最佳一致逼近多项式

$$P_{n-1}(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{n-1} x^{n-1}$$

的里米兹方法如下。

(1) 在区间$[a,b]$上取 n 次切比雪夫(Chebyshev)多项式的交错点组

$$x_k = \frac{1}{2}\left[b + a + (b-a)\cos\frac{n-k}{n}\pi\right], \quad k = 0,1,\cdots,n$$

作为初始参考点集。

(2) 以参考点$\{x_i\}(i=0,1,\cdots,n)$构造一个参考多项式

$$P_{n-1}(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{n-1} x^{n-1}$$

满足

$$P_{n-1}(x_i) - f(x_i) = (-1)^i \mu, \quad i = 0,1,\cdots,n$$

由于 $P_{n-1}(x)$为 $n-1$ 次多项式，在$(n+1)$个点$\{x_i\}(i=0,1,\cdots,n)$上的 n 阶差商为零，由此可以确定出参考偏差 μ。

然后根据 $P_{n-1}(x)$在$(n+1)$个点$\{x_i\}(i=0,1,\cdots,n)$上的各阶差商，利用牛顿插值公式确定出参考多项式 $P_{n-1}(x)$的各系数 $p_0, p_1, \cdots, p_{n-1}$。

(3) 找出使函数$|f(x)-P_{n-1}(x)|$在区间$[a,b]$上取最大值的点 x^*，并按如下原则替换原参考点集$\{x_i\}(i=0,1,\cdots,n)$中的某一点：

① 若 x^* 在 a 与 x_0 之间，且 $f(x_0)-p_{n-1}(x_0)$与 $f(x^*)-P_{n-1}(x^*)$ 同号，则将 x^* 代替 x_0，构成新点集

$$\{x^*, x_1, \cdots, x_n\}$$

否则新集点为

$$\{x^*, x_0, x_1, \cdots, x_{n-1}\}$$

② 若 x^* 在 x_n 与 b 之间，且 $f(x_n)-P_{n-1}(x_n)$ 与 $f(x^*)-P_{n-1}(x^*)$ 同号，则将 x^* 代替 x_n，构成新点集

$$\{x_0, x_1, \cdots, x_{n-1}, x^*\}$$

否则取新点集为 $\{x_1, x_2, \cdots, x_n, x^*\}$。

③ 若 x^* 在 x_i 与 $x_{i+1}(i=0,1,\cdots,n-1)$ 之间，且 $f(x_i)-P_{n-1}(x_i)$ 与 $f(x^*)-P_{n-1}(x^*)$ 同号，则将 x^* 替换 x_i，否则将 x^* 替换 x_{i+1} 构成新点集。

重复(2)与(3)，直到相邻两次求得的参考偏差接近相等为止。此时，最后获得的参考多项式 $P_{n-1}(x)$ 即为近似的 $n-1$ 次最佳一致逼近多项式。

3. 函数原型

```
//最佳一致逼近多项式里米兹法
template <class _Ty>
void ApproximationRemez(_Ty a, _Ty b, valarray<_Ty>& p,_Ty eps);
```

4. 参数

a：_Ty 类型变量。区间左端点值。

b：_Ty 类型变量。区间右端点值。

p：valarray<_Ty>类型引用，长度为 $n+1$。其中 n 个元素返回 $n-1$ 次最佳一致逼近多项式 $P_{n-1}(x)$ 的 n 个系数 $p_0, p_1, \cdots, p_{n-1}$；最后一个元素 $p(n)$ 返回 $P_{n-1}(x)$ 的偏差绝对值 μ。

eps：_Ty 类型变量。控制精度要求。一般取 $10^{-10}\sim10^{-35}$ 之间的数。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 设最佳一致逼近多项式的项数为 $n-1$ 次。要求 $n\leqslant20$。若 $n>20$，本函数将自动取 $n=20$。

(2) 输入、输出的数据应为浮点数类型。

(3) 本函数要调用计算函数 $f(x)$ 函数值的函数 FunctionValueAR()，由用户自己编写，其形式如下：

```
template <class _Ty>
_Ty FunctionValueAR(_Ty x)
{
  _Ty y;
  y = f(x)的表达式;
  return(y);
}
```

7. 示例

函数 $f(x)=e^x$ 在区间[−1,1]上的三次最佳一致逼近多项式及其偏差的绝对值。其中 $a=-1.0, b=1.0, n=4$。取 $eps=10^{-15}$。

主函数程序及计算函数 $f(x)$ 值的函数示例程序如下：

```
//ApproximationRemez.cpp          最佳一致逼近多项式里米兹法
```

```
#include <iostream>                    //模板类输入输出流标准头文件
#include "FittingApproximation.h"      //拟合与逼近头文件
using namespace std;                   //名字空间

void main(void)
{
  valarray<double> p(5);

  double a(-1), b(1), eps(DOUBLEERROR);

  ApproximationRemez(a,b,p,eps);

  for(int i=0; i<4; i++)
      cout << "p[" << i << "] = " << p[i] << endl;

  cout << endl << "MAX(p-f) = " << p[4] << endl;
}

//计算函数值
template <class _Ty>
_Ty FunctionValueAR(_Ty x)
{
  _Ty y = exp(x);
    return(y);
}
```

8. 示例结果

```
p[0] = 0.994594
p[1] = 0.995683
p[2] = 0.542974
p[3] = 0.179518

MAX(p-f) = 0.00551304
```

12.2.4 矩形域的最小二乘曲面拟合

1. 功能

用最小二乘法求矩阵 $n\times m$ 个数据点的拟合曲面。

2. 算法

设已知矩形区域内 $n\times m$ 个网点$(x_i,y_i)(i=0,1,\cdots,n-1;j=0,1,\cdots,m-1)$上的函数值 z_{ij},求最小二乘拟合多项式

$$f(x,y)=\sum_{i=0}^{p-1}\sum_{j=0}^{q-1}a_{ij}x^iy^j$$

首先,固定 y,对 x 构造 m 个最小二乘拟合多项式

$$g_j(x)=\sum_{k=0}^{p-1}\lambda_{kj}\varphi_k(x),j=0,1,\cdots,m-1$$

其中各 $\varphi_k(x)(k=0,1,\cdots,p-1)$ 为互相正交的多项式，并由以下递推公式构造：

$$\varphi_0(x)=1$$

$$\varphi_1(x)=x-\alpha_0$$

$$\varphi_{k+1}(x)=(x-\alpha_k)\varphi_k(x)-\beta_k\varphi_{k-1}(x),k=1,2,\cdots,p-1$$

若令

$$d_k=\sum_{i=0}^{n-1}\varphi_k^2(x_i),k=0,1,\cdots,p-1$$

则有

$$\alpha_k=\sum_{i=0}^{n-1}x_i\varphi_k^2(x_i)/d_k,k=0,1,\cdots,p-1$$

$$\beta_k=d_k/d_{k-1},k=1,2,\cdots,p-1$$

根据最小二乘原理可得

$$\lambda_{kj}=\sum_{i=0}^{n-1}z_{ij}\varphi_k(x_i)/d_k,j=0,1,\cdots,m-1$$

$$k=0,1,\cdots,p-1$$

然后，再构造 y 的最小二乘拟合多项式

$$h_k(y)=\sum_{l=0}^{q-1}\mu_{kl}\Psi_l(y),k=0,1,\cdots,p-1$$

其中 $\Psi_l(y)(l=0,1,\cdots,q-1$ 也为互相正交的多项式，并由以下递推公式构造：

$$\Psi_0(y)=1$$

$$\Psi_1(y)=y-\alpha'_0$$

$$\Psi_{l+1}(y)=(y-\alpha'_l)\Psi_l(y)-\beta'_l\Psi_{l-1}(y),l=1,2,\cdots,q-1$$

若令

$$\delta_l=\sum_{j=0}^{m-1}\Psi_l^2(y_j),l=0,1,\cdots,q-1$$

则有

$$\alpha'_l=\sum_{j=0}^{m-1}y_j\Psi_l^2(y_j)/\delta_l,l=0,1,\cdots,q-1$$

$$\beta'_l=\delta_l/\delta_{l-1},l=1,2,\cdots,q-1$$

根据最小二乘原理可得

$$\mu_{kl}=\sum_{j=0}^{m-1}\lambda_{kj}\Psi_l(y_j)/\delta_l,k=0,1,\cdots,p-1$$

$$l=0,1,\cdots,q-1$$

最后可得二元函数的拟合多项式

$$f(x,y)=\sum_{k=0}^{p-1}\sum_{l=0}^{q-1}\mu_{kl}\varphi_k(x)\Psi_l(y)$$

再转换成标准的多项式形式

$$f(x,y)=\sum_{i=0}^{p-1}\sum_{j=0}^{q-1}a_{ij}x^iy^j$$

在实际计算过程中，为了防止运算溢出 x_i 与 y_i 分别用

$$x_i^* = x_i - \bar{x}, i = 0,1,\cdots,n-1$$

$$y_j^* = y_j - \bar{y}, j = 0,1,\cdots,m-1$$

代替。其中

$$\bar{x} = \sum_{i=0}^{n-1} x_i/n, y = \sum_{j=0}^{m-1} y_j/m$$

此时，二元拟合多项式的形式为

$$f(x,y) = \sum_{i=0}^{p-1}\sum_{j=0}^{q-1} a_{ij}(x-\bar{x})^i(y-\bar{y})^j$$

3. 函数原型

```
//矩形域的最小二乘曲面拟合
template <class _Ty>
void FitSurfaceLeastSquares(valarray<_Ty>& x, valarray<_Ty>& y,matrix<_Ty>& z,
            matrix<_Ty>& a, valarray<_Ty>& dt);
```

4. 参数

x：valarray<_Ty>类型引用，长度为 n，存放给定数据点的 n 个 x 坐标。

y：valarray<_Ty>类型引用，长度为 m，存入给定数据点的 m 个 y 坐标。

z：双精度实型二维数组，体积为 $n\times m$。存放矩形区域内 $n\times m$ 个网点$(x_i,y_j)(i=0,1,\cdots,n-1;j=0,1,\cdots,m-1)$上的函数值 z_{ij}。

a：matrix<_Ty>类型引用，体积为 $p\times q$。返回二元拟合多项式

$$f(x,y) = \sum_{i=0}^{p-1}\sum_{j=0}^{q-1} a_{ij}(x-\bar{x})^i(y-\bar{y})^j$$

的各系数 $a_{ij}(i=0,1,\cdots,p-1;j=0,1,\cdots,q-1)$。

p：整型变量。拟合多项式中 x 的最高次数加 1，即 x 的最高次为 $p-1$。要求 $p\leqslant n$ 且 $p\leqslant 20$。若不满足这个条件，在本函数中自动取 $p=\min\{n,20\}$处理。

q：整型变量。拟合多项式中 y 的最高次数加 1，即 y 的最高次为 $q-1$。要求 $q\leqslant m$ 且 $q\leqslant 20$。若不满足这个条件，在本函数中自动取 $q=\min\{m,20\}$处理。

dt：valarray<_Ty>类型引用，长度为 3。其中：$dt(0)$返回拟合多项式与数据点误差的平方和，即

$$dt(0) = \sum_{i=0}^{n-1}\sum_{j=0}^{m-1}[f(x_i,y_j)-z_{ij}]^2$$

$dt(1)$返回拟合多项式与数据点误差的绝对值之和，即

$$dt(1) = \sum_{i=0}^{n-1}\sum_{j=0}^{m-1}|f(x_i,y_j)-z_{ij}|$$

$dt(2)$返回拟合多项式与数据点误差绝对值的最大值，即

$$dt(2) = \max_{\substack{0\leqslant i\leqslant n-1\\ 0\leqslant j\leqslant m-1}}|f(x,y)-z_{ij}|$$

5. 返回值

本函数无返回值。

6. 函数说明

(1) 二维拟合多项式 $f(x,y)=\sum_{i=0}^{p-1}\sum_{j=0}^{q-1}a_{ij}(x-\bar{x})^i(y-\bar{y})^j$ 中的变量 x 的最高次数为 $p-1$，

并要求 $p \leqslant n$ 且 $p \leqslant 20$。若不满足这个条件，在本函数中自动取 $p=\min\{n,20\}$处理；变量 y 的最高次数为 $q-1$，并要求 $q \leqslant m$ 且 $q \leqslant 20$。若不满足这个条件，在本函数中自动取 $q=\min\{m,20\}$处理。

(2) 输入、输出的数据应为浮点数类型。

7. 示例

设二元函数为 $z(x,y)=e^{x^2\ y^2}$，取矩形区域内 11×21 个网点

$$\begin{cases} x_i = 0.2i, i = 0,1,\cdots,10 \\ y_j = 0.1j, j = 0,1,\cdots,20 \end{cases}$$

上的函数值 $z(x_i,y_j)(i=0,1,\cdots,10;j=0,1,\cdots,20)$。由这些数据构造成一个最小二乘拟合多项式

$$f(x,y) = \sum_{i=0}^{5}\sum_{j=0}^{4} a_{ij}(x-\bar{x})^i(y-\bar{y})^j$$

其中

$$\bar{x} = \sum_{i=0}^{10} x_i/11 = 1.0$$

$$\bar{y} = \sum_{j=0}^{20} y_j/21 = 1.0$$

并分别计算此拟合多项式与各网点上的函数值之差的平方之和 $dt(0)$、绝对值之和 $dt(1)$及绝对值最大者 $dt(2)$。

主函数示例程序如下：

```
//FitSurfaceLeastSquares.cpp        矩形域的最小二乘曲面拟合

#include <iostream>                 //模板类输入输出流标准头文件
#include "FittingApproximation.h"   //拟合与逼近头文件
using namespace std;                //名字空间

void main(void)
{
  int i, j;
  matrix<double> z(11,21), a(6,5);
  valarray<double> x(11), y(21), dt(3);

  for(i=0; i<11; i++) x[i]=0.2*i;

    for(i=0; i<21; i++) y[i]=0.1*i;

  for (i=0; i<11; i++)
     for(j=0; j<21; j++)
       z(i,j)=exp(x[i]*x[i]-y[j]*y[j]);

  cout.setf(ios::fixed);   //输出数据为定点法
  cout.precision(6);       //精度6位
```

```
    FitSurfaceLeastSquares(x,y,z,a,dt);

    cout << "MAT A(i,j) IS: " << endl << endl;

    MatrixLinePrint(a);

    cout << endl << "\t";

    for(i = 0; i<3; i + + )
        cout << "dt(" << i << ") = " << dt[i] << "\t";

    cout << endl;
}
```

8. 示例结果

```
MAT A(i,j) IS:

    1.119238     -2.220087     0.997902     0.738116     -0.581947
    2.252020     -4.467038     2.007878     1.485164     -1.170936
    0.916373     -1.817691     0.817030     0.604331     -0.476468
    1.348363     -2.674571     1.202187     0.889219     -0.701080
    8.197009     -16.259337    7.308372     5.405770     -4.262029
    6.302570     -12.501587    5.619309     4.156424     -3.277017

    dt(0) = 5.801387  dt(1) = 25.376145  dt(2) = 0.555972
```

第 13 章　数据处理与回归分析

本章介绍有关数据处理与回归分析的各种计算方法，其中的每种算法都用 C++语言以函数的形式实现。13.1 节介绍数据处理与回归分析的基本概念，以及数据处理与回归分析算法程序原型的头文件，13.2 节给出了计算各数据处理与回归分析的使用示例及相关说明。

13.1　基本概念与头文件

本节介绍数据处理与回归分析的基本概念，并包括数据处理与回归分析算法的程序原型，其算法与程序原型包含在头文件 Statistic. h 中，函数的定义内容在 Statistic. inl 文件中。

本章介绍的数据处理与回归分析算法及程序包括：

- 随机样本分析；
- 一元线性回归分析；
- n 元线性回归分析；
- 逐步回归分析；
- 半对数数据相关；
- 对数数据相关。

13.1.1　数据处理与回归分析的基本概念

数据处理与回归分析就是利用数学上概率与统计的原理与方法，对在科学研究与工程开发中得到的数据进行分析与处理，以获得科学合理的结果。

13.1.2　Statistic. h

```
//Statistic.h          数据处理与回归分析头文件
// Ver 1.0.0.0
// 版权所有(C) 何渝，2002
// 最后修改：2002.9.6.

#ifndef _STATISTIC_H                    //避免多次编译
#define _STATISTIC_H

#include <iostream>                     //模板类输入输出流 iostream 标准头文件
#include <valarray>                     //模板类 valarray 的标准头文件
#include <Matrix.h>                     //模板类 Matrix 头文件
#include <LinearEquation.h>             //线性方程组头文件
#include <comm.h>                       //公共 comm 头文件
```

```
using namespace std;                    //名字空间

//随机样本分析
template <class _Ty>
void StatRandomSample(valarray<_Ty>& x, _Ty x0, _Ty h, int l,
          valarray<_Ty>& dt, valarray<int>& g, valarray<int>& q);

//一元线性回归分析
template <class _Ty>
void LinearRegression1D(valarray<_Ty>& x, valarray<_Ty>& y,
                              valarray<_Ty>& a, valarray<_Ty>& dt);

//n元线性回归分析
template <class _Ty>
void LinearRegressionND(matrix<_Ty>& x, valarray<_Ty>& y,
          valarray<_Ty>& a, valarray<_Ty>& dt, valarray<_Ty>& v);

//逐步回归分析
template <class _Ty>
void StepwiseRegression(matrix<_Ty>& x, _Ty f1, _Ty f2,
      _Ty eps , valarray<_Ty>& xx, valarray<_Ty>& b, valarray<_Ty>& v,
            valarray<_Ty>& s, valarray<_Ty>& dt, valarray<_Ty>& ye,
                                    valarray<_Ty>& yr, matrix<_Ty>& r);
//半对数数据相关
template <class _Ty>
void HalfLogarithmCorrelation(valarray<_Ty>& x, valarray<_Ty>& y,
                                          _Ty t, valarray<_Ty>& a);

//对数数据相关
template <class _Ty>
void LogarithmCorrelation(valarray<_Ty>& x,
                              valarray<_Ty>& y, valarray<_Ty>& a);

#include "Statistic.inl"          //类及相关函数的定义头文件

#endif                            // _STATISTIC_H
```

13.2 数据处理与回归分析计算方法

13.2.1 随机样本分析

1. 功能

根据给定的一维随机样本,要求:

(1) 计算算术平均值、方差及标准差。

(2) 按高斯分布计算出在各给定区间上近似的理论样本点数。

(3) 打印经验直方图。

2. 算法

设给定随机变量 x 的 n 个样本点值 $x_i(i=0,1,\cdots,n-1)$

(1) 计算样本参数值。

随机样本算术平均值 $\bar{x}=\sum_{i=0}^{n-1}x_i/n$

样本的方差 $s=\sum_{i=0}^{n-1}(x_i-\bar{x})^2/n$

样本的标准差 $t=\sqrt{s}$

(2) 按高斯分布计算出给定各区间上的近似理论样本点数。

设随机变量 x 的起始值为 x_0，区间长度为 h，则第 i 个区间的中点为

$$x_i^*=x_0+(i-0.5)h, i=1,2,\cdots$$

在第 i 个区间上，按高斯分布所应有的近似理论样本点数为

$$F_i=\frac{n}{\sqrt{2\pi s}}\exp\left(-\frac{(x_i^*-\bar{x})^2}{2s}\right)\cdot h$$

(3) 打印经验直方图。

在直方图上方打印样本点数 n，直方图中随机变量的起始值 x_0、等区间长度值 h、区间总数 m，随机变量样本的算术平均值 xa、方差 s 及标准差 t。

在打印的直方图中，左起第一列为从小到大打印各区间的中点值；第二列打印随机样本中落在区间中的实际点数。

右边是直方图本身。各区间对应行上的符号“X”的个数代表样本中随机变量值落在该区间中的点数，而“*”所占的序数则为按高斯分布计算得到的近似理论点数。

在直方图的下方打印直方图的比例 k，即在直方图中，每一个符号表示 k 个点。

3. 函数原型

```
//随机样本分析
template <class _Ty>
void StatRandomSample(valarray<_Ty>& x, _Ty x0, _Ty h, int l,
      valarray<_Ty>& dt, valarray<int>& g, valarray<int>& q);
```

4. 参数

x：valarray<_Ty>类型引用，长度为 n。存放随机变量的 n 个样本点值。

x_0：_Ty 类型变量。直方图中随机变量的起始值。

h：_Ty 类型变量。直方图中随机变量等区间长度值。

l：整型变量。输入标志。若 $l=0$，则不需打印直方图；若 $l\neq0$，则需打印直方图。

dt：valarray<_Ty>类型引用，长度为 3。其中 $dt(0)$ 返回随机样本的算术平均值，$dt(1)$ 返回随机样本的方差 s，$dt(2)$ 返回随机样本的标准差。

g：valarray<int>类型引用，长度为 m。返回 m 个区间的按高斯分布所应有的近似理论样本点数。

q：valarray<int>类型引用，长度为 m。返回落在 m 个区间中的每一个区间上的随机样

本实际点数。

5. 返回值

本函数无返回值。

6. 函数说明

(1)直方图中区间总数为 m。

(2) 除参数 g、q 和 l 外，输入、输出的数据应为浮点数类型。

7. 示例

给定随机变量的 100 个样本点(参看主函数程序)，打印出直方图。其中 $x_0 = 192.000$，$h = 2.00000$，$m = 10$，$l \neq 0$ 。

主函数示例程序如下：

```
//StatRandomSample.cpp        随机样本分析

#include <valarray>           //模板类 valarray 的标准头文件
#include <Statistic.h>        //数据处理与回归分析头文件
using namespace std;          //名字空间

void main(void)
{
      valarray<int> g(10);
      valarray<int> q(10);
      valarray<long double> dt(3);
      long double x0(192), h(2);

      long double x[100] =
      {
          193.199,195.673,195.757,196.051,196.092,196.596,
          196.579,196.763,196.847,197.267,197.392,197.477,
          198.189,193.850,198.944,199.070,199.111,199.153,
          199.237,199.698,199.572,199.614,199.824,199.908,
          200.188,200.160,200.243,200.285,200.453,200.704,
          200.746,200.830,200.872,200.914,200.956,200.998,
          200.998,201.123,201.208,201.333,201.375,201.543,
          201.543,201.584,201.711,201.878,201.919,202.004,
          202.004,202.088,202.172,202.172,202.297,202.339,
          202.381,202.507,202.591,202.716,202.633,202.884,
          203.051,203.052,203.094,203.094,203.177,203.178,
          203.219,203.764,203.765,203.848,203.890,203.974,
          204.184,204.267,204.352,204.352,204.729,205.106,
          205.148,205.231,205.357,205.400,205.483,206.070,
          206.112,206.154,206.155,206.615,206.657,206.993,
          207.243,207.621,208.124,208.375,208.502,208.628,
          208.670,208.711,210.012,211.394
```

```
    };

    valarray<long double> xx(x,100);

    StatRandomSample(xx,x0,h,l,dt,g,q);
}
```

8. 示例结果

```
n = 100

x0 = 192  h = 2  m = 10

xa = 202.23  s = 12.9867  t = 3.60371

193  2  XX
195  2  XX*
197  9  XXXXXXX*X
199  11 XXXXXXXXXXX  *
201  23 XXXXXXXXXXXXXXXXXXXX*XX
203  25 XXXXXXXXXXXXXXXXXXXXX*XXX
205  11 XXXXXXXXXXX  *
207  9  XXXXXXXXX*
209  6  XXX*XX
211  2  X*

k = 1
```

13.2.2 一元线性回归分析

1. 功能

给定 n 个数据点$(x_i,y_i)(i=0,1,\cdots,n-1)$，用直线 $y=ax+b$ 作回归分析。

2. 算法

设随机变量 y 随自变量 x 变化。给定 n 组观测数据$(x_i,y_i)i=0,1,\cdots,n-1$，用直线 $y=ax+b$ 作回归分析。其中 a,b 为回归系数。

为确定回归系数 a,b，通常采用最小二乘法，即要使

$$Q=\sum_{i=0}^{n-1}[y_i-(ax_i+b)]^2$$

达到最小。根据极值原理，a 与 b 满足下列方程

$$\frac{\partial Q}{\partial a}=2\sum_{i=0}^{n-1}[y_i-(ax_i+b)](-x_i)=0$$

$$\frac{\partial Q}{\partial b}=2\sum_{i=0}^{n-1}[y_i-(ax_i+b)](-1)=0$$

从而解得

$$a=\frac{\sum_{i=0}^{n-1}(x_i-\bar{x})(y_i-\bar{y})}{\sum_{i=0}^{n-1}(x_i-\bar{x})^2}$$

$$b=\bar{y}-a\bar{x}$$

其中

$$\bar{x}=\sum_{i=0}^{n-1}x_i/n,\quad \bar{y}=\sum_{i=0}^{n-1}y_i/n$$

最后可以计算出以下几个量：

(1) 偏差平方和 $q=\sum_{i=0}^{n-1}[y_i-(ax_i+b)]^2$

(2) 平均标准偏差 $s=\sqrt{q/n}$

(3) 回归平方和 $p=\sum_{i=0}^{n-1}[(ax_i+b)-\bar{y}]^2$

(4) 最大偏差 $u\max=\max_{0\leqslant i\leqslant n-1}|y_i-(ax_i+b)|$

(5) 最小偏差 $u\min=\min_{0\leqslant i\leqslant n-1}|y_i-(ax_i+b)|$

(6) 偏差平均值 $u=\sum_{i=0}^{n-1}|y_i-(ax_i+b)|/n$

3. 函数原型

```
//一元线性回归分析
template <class _Ty>
void LinearRegression1D(valarray<_Ty>& x, valarray<_Ty>& y,
                        valarray<_Ty>& a, valarray<_Ty>& dt);
```

4. 参数

x:valarray<_Ty>类型引用,长度为 n。存放自变量 x 的 n 个取值。

y:valarray<_Ty>类型引用,长度为 n。存放与自变量 x 的 n 个取值相对应的随机变量 y 的观测值。

a:valarray<_Ty>类型引用,长度为 2。其中 $a(0)$返回回归系数 b,$a(1)$返回回归系数 a。

dt:valarray<_Ty>类型引用,长度为 6。其中 $dt(0)$返回偏差平方和 q,$dt(1)$返回平均标准偏差 s,$dt(2)$返回回归平均和 p,$dt(3)$返回最大偏差 $u\max$,$dt(4)$返回最小偏差 $u\min$,$dt(5)$返回偏差平均值 u。

5. 返回值

本函数无返回值。

6. 函数说明

输入、输出的数据应为浮点数类型。

7. 示例

给定如下 11 个观测值。

X	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Y	2.75	2.84	2.965	3.01	3.20	3.25	3.28	3.43	3.55	3.66	3.74

求回归系数 a 与 b 偏差平方和 q、平均标准偏差 s、回归平均和 p、偏差最大值 $u\max$、偏差最小值 $u\min$ 和偏差平均值 u。

主函数示例程序如下：

```
//LinearRegression1D.cpp 一元线性回归分析

#include <iostream>          //模板类输入、输出流 iostream 标准头文件
#include <valarray>          //模板类 valarray 的标准头文件
#include <Statistic.h>       //数据处理与回归分析头文件
using namespace std;         //名字空间

void main(void)
{
 valarray<long double> a(2);
 valarray<long double> dt(6);
 long double x0(192), h(2);

 long double x[11] =
 {
    0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0
 };

 long double y[11] =
 {
    2.75,2.84,2.965,3.01,3.20,3.25,3.38,3.43,3.55,3.66,3.74
 };

 valarray<long double> xx(x, 11);
 valarray<long double> yy(y, 11);

  LinearRegression1D(xx, yy, a, dt);

 cout << endl << "a = " << a[1] << "\t b = " << a[0] << endl;
 cout << endl << "q = " << dt[0] << "\t s = " << dt[1];
                      cout << "\t p = " << dt[2] << endl;
 cout << endl << "umax = " << dt[3] << "\t umin = " << dt[4];
                                cout << "\t u = " << dt[5] << endl;
}
```

8. 示例结果

```
a = 1.00045     b = 2.75205

q = 0.00586795      s = 0.0230965      p = 1.101

umax = 0.0477727     umin = 0.00204545     u = 0.0174298
```

13.2.3 n 元线性回归分析

1. 功能

根据随机变量 y 及 $x_0,x_1,\cdots,x_{m-1}$ 的 n 组观测值 $(x_{0i},x_{1i},\cdots,x_{m-1,i},y_i)(i=0,1,\cdots,n-1)$

作线性回归分析。

2. 算法

设随机变量 y 及 m 个自变量 $x_0,x_1,\cdots,x_{m-1}$。给定 n 组观测数据 $(x_{0i},x_{1i},\cdots,x_{m-1,i},y_i)$ $(i=0,1,\cdots,n-1)$，用线性表达式

$$y=a_0x_0+a_1x_1+\cdots+a_{m-1}x_{m-1}+a_m$$

对观测数据进行回归分析。其中 $a_0,a_1,\cdots,a_{m-1},a_m$ 为回归系数。

与一元线性回归分析一样(见本章 13.2.2 节的算法说明)，根据最小二乘原理，为使

$$q=\sum_{i=0}^{n-1}[y_i-(a_0x_{0i}+a_1x_{1i}+\cdots+a_{m-1}x_{m-1,i}+a_m)]^2$$

达到最小，回归系数 $a_0,a_1,\cdots,a_{m-1},a_m$ 应满足下列方程组

$$(CC^{\mathrm{T}})\begin{bmatrix}a_0\\a_1\\a_2\\\vdots\\a_{m-1}\\a_m\end{bmatrix}=C\begin{bmatrix}y_0\\y_1\\y_2\\\vdots\\y_{n-21}\\y_{n-1}\end{bmatrix}$$

其中

$$C=\begin{bmatrix}x_{00}&x_{01}&x_{02}&\cdots&x_{0,n-1}\\x_{10}&x_{11}&x_{12}&\cdots&x_{1,n-1}\\\vdots&\vdots&&\vdots&\vdots\\x_{m-1,0}&x_{m-1,1}&x_{m-1,2}&\cdots&x_{m-1,n-1}\\1&1&1&\cdots&1\end{bmatrix}$$

采用乔里斯基(Cholesky)分解法解出回归系数。

为了衡量回归效果，还要计算以下 5 个量。

(1) 偏差平方和 $q=\sum_{i=0}^{n-1}[y_i-(a_0x_{0i}+a_1x_{1i}+\cdots+a_{m-1}x_{m-1,i}+a_m)]^2$

(2) 平均标准偏差 $s=\sqrt{q/n}$

(3) 复相关系数 $r=\sqrt{1-q/t}$

其中

$$t=\sum_{i=0}^{n-1}(y_i-\bar{y})^2,\quad \bar{y}=\sum_{i=0}^{n-1}y_i/n$$

当 r 接近于 1 时，说明相对误差 q/t 接近于零，线性回归效果好。

(4) 偏相关系数 $v_j=\sqrt{1-q/q_j},\quad j=0,1,\cdots,m-1$

其中

$$q_j=\sum_{i=0}^{n-1}[y_i-(a_m+\sum_{\substack{k=0\\k\neq j}}^{m-1}a_kx_{ki})]^2$$

当 v_j 越大时，说明 x_j 对于 y 的作用越显著，此时不可把 x_j 剔除。

(5) 回归平方和 $u=\sum_{i=0}^{n-1}[\bar{y}-(a_0i_{0i}+a_1i_{1i}+\cdots+a_{m-1}x_{m-1,i}+a_m)]^2$

3. 函数原型

```
//n元线性回归分析
template <class _Ty>
void LinearRegressionND(matrix<_Ty>& x, valarray<_Ty>& y,
        valarray<_Ty>& a, valarray<_Ty>& dt, valarray<_Ty>& v);
```

4. 参数

x：_Ty类型二维数组，体积为 $m\times n$。每一列存放 m 个自变量的一组观测值，即

$$(x_{0i},x_{1i},\cdots,x_{m-1,i})^{\mathrm{T}},i=0,1,\cdots,n-1$$

y：valarray<_Ty>类型引用，长度为 n。存放随机变量 y 的 n 个观测值。

a：valarray<_Ty>类型引用，长度为 $m+1$，返回回归系数 $a_0,a_1,\cdots,a_{m-1},a_m$。

dt：valarray<_Ty>类型引用，长度为4，其中 $dt(0)$ 返回偏差平方和 q，$dt(1)$ 返回平均标准偏差 s，$dt(2)$ 返回复相关系数 r，$dt(3)$ 返回回归平方和 u。

v：valarray<_Ty>类型引用，长度为 m。返回 m 个自变量的偏相关系数。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 函数中自变量的个数为 n，观测数据的组数为 m。

(2) 输入、输出的数据应为浮点数类型。

(3) 本函数要调用平方根法求解对称正定方程组的函数 LE_SymmetryRegularEuationSquareRoot()，参看7.2.6节。

7. 示例

对随机变量 y 及自变量 x_0、x_1 和 x_2 的下列5组观测数据作多元线性回归分析 i。

i	x_{0i}	x_{1i}	x_{2i}	y_i
0	1.1	2.0	3.2	10.1
1	1.0	2.0	3.2	10.2
2	1.2	1.8	3.0	10.0
3	1.1	1.9	2.9	10.1
4	0.9	2.1	2.9	10.0

主函数示例程序如下：

```
//LinearRegressionND.cpp        n元线性回归分析

#include <iostream>             //模板类输入、输出流 iostream 标准头文件
#include <valarray>             //模板类 valarray 的标准头文件
#include <Matrix.h>             //模板类 Matrix 头文件
#include <Statistic.h>          //数据处理与回归分析头文件
using namespace std;            //名字空间

void main(void)
{
  valarray<long double> a(4);
  valarray<long double> v(3);
```

```
    valarray<long double> dt(4);
    long double x0(192), h(2);

    long double x[3][5] =
    {
        {1.1,1.0,1.2,1.1,0.9},
         {2.0,2.0,1.8,1.9,2.1},
        {3.2,3.2,3.0,2.9,2.9}
    };

    long double y[5] = {10.1,10.2,10.0,10.1,10.0};

    matrix<long double> xx(&x[0][0], 3, 5);
    valarray<long double> yy(y, 5);

     LinearRegressionND(xx,yy,a,dt,v);

    cout << endl;

    for (int i = 0; i<4; i + + )
        cout << " a(" << i << ") = " << a[i] << endl;

    cout << endl << " q = " << dt[0] << "\t s = " << dt[1];
                                    cout << "\t r = " << dt[2] << endl;

    cout << endl;

    for (i = 0; i<3; i + + )
        cout << " v(" << i << ") = " << v[i] << endl;

    cout << endl;

    cout << " u = " << dt[3] << endl << endl;
}
```

8. 示例结果

```
K = 1    Determinant = 5.67
K = 2    Determinant = 2.2886
K = 3    Determinant = 0.155316
K = 4    Determinant = 0.00012

a(0) = -0.8
a(1) = -0.7
a(2) = 0.5
a(3) = 10.78
```

q = 0.012　　s = 0.0489898　　r = 0.755929

v(0) = 0.998351

v(1) = 0.999365

v(2) = 0.999482

u = 0.016

13.2.4 逐步回归分析

1. 功能

对多元线性回归进行因子筛选，最后给出一定显著性水平下各因子均为显著的回归方程中的诸回归系数、偏回归平方和、估计的标准偏差、复相关系数及 F 一检验值、各回归系数的标准偏差、应变量条件期望值的估计值及残差。

2. 算法

设 n 个自变量为 $x_i(i=0,1,\cdots,n-1)$，因变量为 y。有 k 个观测点为

$$(x_{i0},x_{i1},\cdots,x_{i,n-1},y_i)i=0,1,\cdots,k-1$$

根据最小二乘原理，y 的估计值为

$$\hat{y}=b_{i_0}x_{i_0}+b_{i_1}x_{i_1}+\cdots+b_{i_l}x_{i_l}+b_n$$

其中，$0\leqslant i_0\leqslant i_1\cdots\leqslant i_l\leqslant n-1$，且各 $x_{i_t}(t=0,1,\cdots,l)$ 是从 n 个自变量 $x_j(i=0,1,\cdots,n-1)$ 中按一定显著性水平筛选出的统计检验为显著的因子。其筛选过程如下。

(1) 作出 $(n+1)\times(n+1)$ 的规格化系数初始相关阵

$$R=\begin{bmatrix} r_{00} & r_{01} & \cdots & r_{0,n-1} & r_{0y} \\ r_{10} & r_{11} & \cdots & r_{1,n-1} & r_{1y} \\ \vdots & \vdots & & \vdots & \vdots \\ r_{n-1,0} & r_{n-1,1} & \cdots & r_{n-1,n-1} & r_{n-1,y} \\ r_{y0} & r_{y1} & \cdots & r_{y,n-1} & r_{yy} \end{bmatrix}$$

矩阵中各元素为

$$r_{ij}=\frac{d_{ij}}{d_id_j}=\frac{\sum_{t=0}^{k-1}(x_{ti}-\bar{x}_i)(x_{tj}-\bar{x}_j)}{\sqrt{\sum_{t=0}^{k-1}(x_{ti}-\bar{x}_i)^2}\cdot\sqrt{\sum_{t=0}^{k-1}(x_{tj}-\bar{x}_j)^2}}$$

$$i,j=0,1,\cdots,n-1,n$$

其中 n 对应 y。式中

$$\bar{x}_i=\sum_{t=0}^{k-1}x_{ti}/k,i=0,1,\cdots,n-1,n$$

(2) 计算偏回归平方和　　$V_i=\frac{r_{iy}r_{yi}}{r_{ii}},i=0,1,\cdots,n-1$

(3) 若 $V_i<0$，则对应 x_i 已是被选入回归方程的因子。

从所有 $V_i<0$ 的 V_i 中选出 $V_{\min}=\min|V_i|$，其对应的因子为 $x_{\min}$，然后检验因子 $x_{\min}$ 的显著性。若

$$\frac{\varphi V_{\min}}{r_{yy}} < F_2$$

则剔除因子 $x_{\min}$，并对系数相关阵 $\boldsymbol{R}$ 进行该因子的消元变换。转(2)。

(4) 若 $V_i > 0$，则对应的 x_i 为尚待选入的因子。

从所有 $V_i > 0$ 的 V_i 中选出 $V_{\max} = \max|V_i|$，其对应的因子为 $x_{\max}$，然后检验因子 $x_{\max}$ 的显著性。若

$$\frac{(\varphi-1)V_{\max}}{r_{yy}-V_{\max}} \geqslant F_1$$

则因子 $x_{\max}$ 应选入，并对系数相关阵 $\boldsymbol{R}$ 进行该因子的消元变换。转(2)。

上述过程一直进行到没有因子可剔除或可选入为止。

在以上步骤中，φ 为相应的残差平方和的自由度。F_1 及 F_2 均是 F 一分布，它们取决于观测点数、已选入的因子数及取舍显著性水平 α。通常取 $F_1 > F_2$，当选入单个因子的显著性水平取为 α 时，则可以从 F 一分布表中取 $m=1$，观测点数为 n 时的 F_α 为 F_2，而取 $m=1$，观测点数为 $n=1$ 时的 F_α 为 F_1。

当要剔除或选入某个因子 x_l 时，均需对系数相关阵 $\boldsymbol{R}$ 进行消元变换，其算法如下：

$$r_{ij} = r_{ij} - \frac{r_{il}}{r_{ll}} r_{lj}, i,j = 0,1,\cdots,n; i,j \neq l$$

$$r_{lj} = r_{lj}/r_{ll}, \quad j = 0,1,\cdots,n; j \neq l$$

$$r_{il} = -r_{il}/r_{ll} \quad i = 0,1,\cdots,n; i \neq l$$

$$r_{ll} = 1/r_{ll}$$

当筛选结束时，就可得出规格化回归方程的各回归系数 $b_0, b_1, b_2, \cdots, b_n$，其中值为 0 的系数表示对应的自变量可剔除。

回归模型的各有关值可由下列各式计算：

- 选入回归方程的各因子的回归系数 $\quad b_i = \dfrac{d_y}{d_i} r_{iy}, i = 0,1,\cdots,n-1$
- 回归方程的常数项 $\quad b_n = \bar{y} - \sum_{i=0}^{n-1} b_j \bar{x}_j$
- 各因子的偏差回归平方和 $\quad V_i = r_{iy} r_{yi}/r_{ii}, i = 0,1,\cdots,n-1$
- 估计的标准偏差 $\quad s = d_y \sqrt{r_{yy}/\varphi}$
- 各回归系数的标准偏差 $\quad s_i = s\sqrt{r_{ii}}/d_i, i = 0,1,\cdots,n-1$
- 复相关系数 $\quad C = \sqrt{1-r_{yy}}$
- F一检验值 $\quad F = \dfrac{\varphi(1-r_{yy})}{(k-\varphi-1)r_{yy}}$
- 残差平方和 $\quad q = d_y^2 r_{yy}$
- 应变量条件期望值的估计值 $\quad e_i = b_n + \sum_{j=0}^{n-1} b_j x_{ij}, i = 0,1,\cdots,k-1$
- 残值 $\quad \delta_i = y_i - e_i, i = 0,1,\cdots,k-1$

本函数适用于自变量个数较多，且观测点较多的问题。

3. 函数原型

```
//逐步回归分析
template <class _Ty>
```

```
void StepwiseRegression(matrix<_Ty>& x, _Ty f1, _Ty f2,
    _Ty eps , valarray<_Ty>& xx, valarray<_Ty>& b, valarray<_Ty>& v,
        valarray<_Ty>& s, valarray<_Ty>& dt, valarray<_Ty>& ye,
                    valarray<_Ty>& yr, matrix<_Ty>& r);
```

4. 参数

x:matrix<_Ty>类型引用,体积为 $k\times(n+1)$。其中前 n 列存入自变量因子 $x_i(i=0,1,\cdots,n-1)$ 的 k 次观测,最后一列存放应变量 y 的 k 次观测值。

f1:_Ty 类型变量。置欲选入因子时显著性检验的 F一分布值。

f2:_Ty 类型变量。置欲剔出因子时显著性检验的 F一分布值。

eps:_Ty 类型变量。置防止系数相关阵退化的判据。

xx:valarray<_Ty>类型引用,长度为 $n+1$。其中前 n 个分量返回 n 个自变量因子的算术平均值 $\bar{x}_i(i=0,1,\cdots,n-1)$,第 $n+1$ 个分量(即 $xx(n)$)返回应变量 y 的算术平均值 $\bar{y}$。

b:valarray<_Ty>类型引用,长度为 $n+1$。返回回归方程中各因子的回归系数及常数项 $b_0,b_1,\cdots,b_n$。

v:valarray<_Ty>类型引用,长度为 $n+1$。其中前 n 个分量返回各因子的偏回归平方和 $v_i(i=0,1,\cdots,n-1)$,最后一个分量(即 $v(n)$)返回残差平方和 q。

s:valarray<_Ty>类型引用,长度为 $n+1$。其中前 n 个分量返回各因子回归系数的标准偏差 $s_i(i=0,1,\cdots,n-1)$,最后一个分量(即 $s(n)$)返回残差平方和估计的标准偏差 s。

dt:valarray<_Ty>类型引用,长度为 2。其中 $dt(0)$返回复相关系数 c,$dt(1)$返回 F一检验值。

ye:valarray<_Ty>类型引用,长度为 k。返回因变量条件期望值的 k 个估计值(对应于 k 个观测值) $e_i(i=0,1,\cdots,k-1)$。

yr:valarray<_Ty>类型引用,长度为 k。返回因变量的 k 个观测值的残差 $\delta_i(i=0,1,\cdots,k-1)$。

r:matrix<_Ty>类型引用,体积为 $(n+1)\times(n+1)$。返回最终的规格化的系数相关阵 $\boldsymbol{R}$。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数中设自变量 x 的个数为 n,观测点数为 k。

(2) 输入、输出的数据应为浮点数类型。

7. 示例

设 4 个自变量为 x_0、x_1、x_2 和 x_3,因变量为 y。13 个观测点如下:

k	x_0	x_1	x_2	x_3	y
0	7.0	26.0	6.0	60.0	78.5
1	1.0	29.0	15.0	52.0	74.3
2	11.0	56.0	8.0	20.0	104.3
3	11.0	31.0	8.0	47.0	87.6
4	7.0	52.0	6.0	33.0	95.9
5	11.0	55.0	9.0	22.0	109.2
6	3.0	71.0	17.0	6.0	102.7

续表

k	x_0	x_1	x_2	x_3	y
7	1.0	31.0	22.0	44.0	72.5
8	2.0	54.0	18.0	22.0	93.1
9	21.0	47.0	4.0	26.0	115.9
10	1.0	40.0	23.0	34.0	83.8
11	11.0	66.0	9.0	12.0	113.3
12	10.0	68.0	8.0	12.0	109.4

对于不同的 F_1 与 F_2 值进行逐步回归分析。

(1) 当取 $\alpha = 0.25$ 时，查 F一分布表得 $F_1 = 1.46, F_2 = 1.45$。

(2) 当取 $\alpha = 0.05$ 时，查 F一分布表得 $F_1 = 4.75, F_2 = 4.67$。

(3) 当取 $\alpha = 0.01$ 时，查 F一分布表得 $F_1 = 9.33, F_2 = 9.07$。

主函数示例程序如下：

```
//StepwiseRegression.cpp 逐步回归分析

#include <iostream>          //模板类输入、输出流 iostream 标准头文件
#include <valarray>          //模板类 valarray 的标准头文件
#include <Statistic.h>       //数据处理与回归分析头文件
#include <Matrix.h>          //模板类 Matrix 头文件
using namespace std;         //名字空间

void main(void)
{
  valarray<long double> dt(2),xx(5),b(5),v(5),s(5),ye(13),yr(13);
  long double eps = LONGDOUBLEERROR;
  matrix<long double> r(5,5);

  long double x[13][5] =
  {
       {7.0,26.0,6.0,60.0,78.5}, {1.0,29.0,15.0,52.0,74.3},
        {11.0,56.0,8.0,20.0,104.3}, {11.0,31.0,8.0,47.0,87.6},
        {7.0,52.0,6.0,33.0,95.9}, {11.0,55.0,9.0,22.0,109.2},
       {3.0,71.0,17.0,6.0,102.7}, {1.0,31.0,22.0,44.0,72.5},
       {2.0,54.0,18.0,22.0,93.1}, {21.0,47.0,4.0,26.0,115.9},
       {1.0,40.0,23.0,34.0,83.8}, {11.0,66.0,9.0,12.0,113.3},
        {10.0,68.0,8.0,12.0,109.4}
  };

  matrix<long double> xx2(&x[0][0],13,5);

  long double f1[3] = {1.46,4.75,9.33};
  long double f2[3] = {1.45,4.67,9.07};

  for(int k = 0; k<3; k++)
```

```
{
    StepwiseRegression(xx2,f1[k],f2[k],eps,xx,b,v,s,dt,ye,yr,r);

    printf("\n");
     printf("f1 = %e f2 = %e\n",f1[k],f2[k]);
     printf("\n");
     printf("original x(i) and y values:\n");
     for(int i = 0; i<13; i++)
     {
       for(int j = 0; j<=3; j++)printf("x(%d) = %6.2f ",j,x[i][j]);
       printf("y= %6.2f\n",x[i][4]);
     }
     printf("\n");
     printf("mean of x(i) and y:\n");
     for(i = 0; i<4; i++)
       printf("x(%d) = %6.3f ",i,xx[i]);
     printf("y= %6.3f\n",xx[4]);
     printf("\n");
     printf("regression coeffi b(i):\n");
     for(i = 0; i<=4; i++)
       printf("b(%d) = %9.6f ",i,b[i]);
     printf("\n\n");
     printf("standard partial sum of square of \n");
     printf("regression for x(i) and sum of \n");
     printf("square of residuals:\n");
     for(i = 0; i<4; i++)
       printf("v(%d) = %e ",i,v[i]);
     printf("\n");
     printf("q= %e\n",v[4]);
     printf("\n");
     printf("standard deviation of regression \n");
     printf("coeffi and regression equation:\n");
     for(i = 0; i<4; i++)
       printf("s(%d) = %e ",i,s[i]);
     printf("\n");
     printf("s = %e\n",s[4]);
     printf("\n");
     printf("multi-correlation coeffi c is: %e\n",dt[0]);
     printf("\n");
     printf("the f value = %e\n",dt[1]);
     printf("\n");
     printf("estimated values and residuals:\n");
     for(i = 0; i<13; i++)
       printf("ye(%d) = %e yr(%d) = %e\n",
```

```
                i,ye[i],i,yr[i]);
        printf("\n");
        printf("matrix r:\n");
        for(i=0; i<5; i++)
        {
            for(int j=0; j<5; j++) printf("%e ",r(i,j));
            printf("\n");
        }
        printf("\n");
    }
}
```

8. 示例结果

```
f1 = 1.460000e+000      f2 = 1.450000e+000

original x(i) and y values:
x(0) = 7.00  x(1) = 26.00  x(2) = 6.00  x(3) = 60.00  y = 78.50
x(0) = 1.00  x(1) = 29.00  x(2) = 15.00 x(3) = 52.00  y = 74.30
x(0) = 11.00 x(1) = 56.00  x(2) = 8.00  x(3) = 20.00  y = 104.30
x(0) = 11.00 x(1) = 31.00  x(2) = 8.00  x(3) = 47.00  y = 87.60
x(0) = 7.00  x(1) = 52.00  x(2) = 6.00  x(3) = 33.00  y = 95.90
x(0) = 11.00 x(1) = 55.00  x(2) = 9.00  x(3) = 22.00  y = 109.20
x(0) = 3.00  x(1) = 71.00  x(2) = 17.00 x(3) = 6.00   y = 102.70
x(0) = 1.00  x(1) = 31.00  x(2) = 22.00 x(3) = 44.00  y = 72.50
x(0) = 2.00  x(1) = 54.00  x(2) = 18.00 x(3) = 22.00  y = 93.10
x(0) = 21.00 x(1) = 47.00  x(2) = 4.00  x(3) = 26.00  y = 115.90
x(0) = 1.00  x(1) = 40.00  x(2) = 23.00 x(3) = 34.00  y = 83.80
x(0) = 11.00 x(1) = 66.00  x(2) = 9.00  x(3) = 12.00  y = 113.30
x(0) = 10.00 x(1) = 68.00  x(2) = 8.00  x(3) = 12.00  y = 109.40

mean of x(i) and y:
x(0) = 7.462 x(1) = 48.154  x(2) = 11.769 x(3) = 30.000  y = 95.423

regression coeffi b(i):
b(0) = 1.451938 b(1) = 0.416110 b(2) = 0.000000 b(3) = -0.236540 b(4) = 71.648307

standard partial sum of square of
regression for x(i) and sum of
square of residuals:
v(0) = -3.022750e-001 v(1) = -9.864403e-003 v(2) = 4.016921e-005 v(3) = -3.657077e-003
q = 4.797273e+001

standard deviation of regression
coeffi and regression equation:
s(0) = 1.169976e-001 s(1) = 1.856105e-001 s(2) = 0.000000e+000 s(3) = 1.732878e-001
```

s = 2. 308745e + 000

multi - correlation coeffi c is:9. 911284e - 001

the f value = 1. 668317e + 002

estimated values and residuals:

ye(0) = 7. 843831e + 001 yr(0) = 6. 168641e - 002
ye(1) = 7. 286734e + 001 yr(1) = 1. 432663e + 000
ye(2) = 1. 061910e + 002 yr(2) = - 1. 890967e + 000
ye(3) = 8. 940164e + 001 yr(3) = - 1. 801637e + 000
ye(4) = 9. 564375e + 001 yr(4) = 2. 562468e - 001
ye(5) = 1. 053018e + 002 yr(5) = 3. 898223e + 000
ye(6) = 1. 041287e + 002 yr(6) = - 1. 428673e + 000
ye(7) = 7. 559188e + 001 yr(7) = - 3. 091878e + 000
ye(8) = 9. 181823e + 001 yr(8) = 1. 281775e + 000
ye(9) = 1. 155461e + 002 yr(9) = 3. 538826e - 001
ye(10) = 8. 170227e + 001 yr(10) = 2. 097732e + 000
ye(11) = 1. 122444e + 002 yr(11) = 1. 055614e + 000
ye(12) = 1. 116247e + 002 yr(12) = - 2. 224668e + 000

matrix r:

1. 066330e + 000 2. 043901e - 001 - 8. 936536e - 001 4. 605878e - 001 5. 677366e - 001
2. 043901e - 001 1. 878031e + 001 - 2. 242266e + 000 1. 832256e + 001 4. 304144e - 001
8. 936536e - 001 2. 242266e + 000 2. 133634e - 002 2. 371430e + 000 9. 257775e - 004
4. 605878e - 001 1. 832256e + 001 - 2. 371430e + 000 1. 894008e + 001 - 2. 631830e - 001
- 5. 677366e - 001 - 4. 304144e - 001 9. 257775e - 004 2. 631830e - 001 1. 766455e - 002

f1 = 4. 750000e + 000 f2 = 4. 670000e + 000

original x(i) and y values:

x(0) = 7. 00 x(1) = 26. 00 x(2) = 6. 00 x(3) = 60. 00 y = 78. 50
x(0) = 1. 00 x(1) = 29. 00 x(2) = 15. 00 x(3) = 52. 00 y = 74. 30
x(0) = 11. 00 x(1) = 56. 00 x(2) = 8. 00 x(3) = 20. 00 y = 104. 30
x(0) = 11. 00 x(1) = 31. 00 x(2) = 8. 00 x(3) = 47. 00 y = 87. 60
x(0) = 7. 00 x(1) = 52. 00 x(2) = 6. 00 x(3) = 33. 00 y = 95. 90
x(0) = 11. 00 x(1) = 55. 00 x(2) = 9. 00 x(3) = 22. 00 y = 109. 20
x(0) = 3. 00 x(1) = 71. 00 x(2) = 17. 00 x(3) = 6. 00 y = 102. 70
x(0) = 1. 00 x(1) = 31. 00 x(2) = 22. 00 x(3) = 44. 00 y = 72. 50
x(0) = 2. 00 x(1) = 54. 00 x(2) = 18. 00 x(3) = 22. 00 y = 93. 10
x(0) = 21. 00 x(1) = 47. 00 x(2) = 4. 00 x(3) = 26. 00 y = 115. 90
x(0) = 1. 00 x(1) = 40. 00 x(2) = 23. 00 x(3) = 34. 00 y = 83. 80
x(0) = 11. 00 x(1) = 66. 00 x(2) = 9. 00 x(3) = 12. 00 y = 113. 30
x(0) = 10. 00 x(1) = 68. 00 x(2) = 8. 00 x(3) = 12. 00 y = 109. 40

mean of x(i) and y:

x(0) = 7.462 x(1) = 48.154 x(2) = 11.769 x(3) = 30.000 y = 95.423

regression coeffi b(i):

b(0) = 1.468306 b(1) = 0.662250 b(2) = 0.000000 b(3) = 0.000000 b(4) = 52.577349

standard partial sum of square of
regression for x(i) and sum of
square of residuals:

v(0) = -3.124101e-001 v(1) = -4.447304e-001 v(2) = 3.606305e-003 v(3) = 3.657077e-003
q = 5.790448e+001

standard deviation of regression
coeffi and regression equation:

s(0) = 1.213009e-001 s(1) = 4.585472e-002 s(2) = 0.000000e+000 s(3) = 0.000000e+000
s = 2.406335e+000

multi-correlation coeffi c is:9.892817e-001

the f value = 2.295037e+002

estimated values and residuals:

ye(0) = 8.007400e+001 yr(0) = -1.574002e+000
ye(1) = 7.325092e+001 yr(1) = 1.049081e+000
ye(2) = 1.058147e+002 yr(2) = -1.514740e+000
ye(3) = 8.925848e+001 yr(3) = -1.658477e+000
ye(4) = 9.729251e+001 yr(4) = -1.392515e+000
ye(5) = 1.051525e+002 yr(5) = 4.047511e+000
ye(6) = 1.040021e+002 yr(6) = -1.302051e+000
ye(7) = 7.457542e+001 yr(7) = -2.075420e+000
ye(8) = 9.127549e+001 yr(8) = 1.824513e+000
ye(9) = 1.145375e+002 yr(9) = 1.362457e+000
ye(10) = 8.053567e+001 yr(10) = 3.264326e+000
ye(11) = 1.124372e+002 yr(11) = 8.627555e-001
ye(12) = 1.122934e+002 yr(12) = -2.893440e+000

matrix r:

1.055129e+000 -2.411808e-001 -8.359848e-001 -2.431816e-002 5.741367e-001
-2.411808e-001 1.055129e+000 5.184659e-002 -9.673964e-001 6.850167e-001
8.359848e-001 -5.184659e-002 3.182559e-001 -1.252070e-001 3.387813e-002
2.431816e-002 9.673964e-001 -1.252070e-001 5.279810e-002 -1.389556e-002
-5.741367e-001 -6.850167e-001 3.387813e-002 -1.389556e-002 2.132163e-002

f1 = 9.330000e + 000 f2 = 9.070000e + 000

original x(i) and y values:

x(0) = 7.00 x(1) = 26.00 x(2) = 6.00 x(3) = 60.00 y = 78.50

x(0) = 1.00 x(1) = 29.00 x(2) = 15.00 x(3) = 52.00 y = 74.30

x(0) = 11.00 x(1) = 56.00 x(2) = 8.00 x(3) = 20.00 y = 104.30

x(0) = 11.00 x(1) = 31.00 x(2) = 8.00 x(3) = 47.00 y = 87.60

x(0) = 7.00 x(1) = 52.00 x(2) = 6.00 x(3) = 33.00 y = 95.90

x(0) = 11.00 x(1) = 55.00 x(2) = 9.00 x(3) = 22.00 y = 109.20

x(0) = 3.00 x(1) = 71.00 x(2) = 17.00 x(3) = 6.00 y = 102.70

x(0) = 1.00 x(1) = 31.00 x(2) = 22.00 x(3) = 44.00 y = 72.50

x(0) = 2.00 x(1) = 54.00 x(2) = 18.00 x(3) = 22.00 y = 93.10

x(0) = 21.00 x(1) = 47.00 x(2) = 4.00 x(3) = 26.00 y = 115.90

x(0) = 1.00 x(1) = 40.00 x(2) = 23.00 x(3) = 34.00 y = 83.80

x(0) = 11.00 x(1) = 66.00 x(2) = 9.00 x(3) = 12.00 y = 113.30

x(0) = 10.00 x(1) = 68.00 x(2) = 8.00 x(3) = 12.00 y = 109.40

mean of x(i) and y:

x(0) = 7.462 x(1) = 48.154 x(2) = 11.769 x(3) = 30.000 y = 95.423

regression coeffi b(i):

b(0) = 1.439958 b(1) = 0.000000 b(2) = 0.000000 b(3) = − 0.613954 b(4) = 103.097382

standard partial sum of square of
regression for x(i) and sum of
square of residuals:

v(0) = − 2.979291e − 001 v(1) = 9.864403e − 003 v(2) = 8.810045e − 003 v(3) = − 4.385230e − 001
q = 7.476211e + 001

standard deviation of regression
coeffi and regression equation:

s(0) = 1.384166e − 001 s(1) = 0.000000e + 000 s(2) = 0.000000e + 000 s(3) = 4.864455e − 002
s = 2.734266e + 000

multi − correlation coeffi c is:9.861395e − 001

the f value = 1.766270e + 002

estimated values and residuals:

ye(0) = 7.633987e + 001 yr(0) = 2.160128e + 000

ye(1) = 7.261175e + 001 yr(1) = 1.688249e + 000

ye(2) = 1.066579e + 002 yr(2) = − 2.357850e + 000

ye(3) = 9.008110e + 001 yr(3) = − 2.481102e + 000

ye(4) = 9.291662e + 001 yr(4) = 2.983380e + 000

```
ye(5) = 1.054299e+002   yr(5) = 3.770057e+000
ye(6) = 1.037335e+002   yr(6) = -1.033535e+000
ye(7) = 7.752338e+001   yr(7) = -5.023380e+000
ye(8) = 9.247032e+001   yr(8) = 6.296816e-001
ye(9) = 1.173737e+002   yr(9) = -1.473711e+000
ye(10) = 8.366292e+001  yr(10) = 1.370834e-001
ye(11) = 1.115695e+002  yr(11) = 1.730521e+000
ye(12) = 1.101295e+002  yr(12) = -7.295210e-001

matrix r:
1.064105e+000 -1.088321e-002 -8.692506e-001 2.611794e-001 5.630523e-001
1.088321e-002 5.324726e-002 -1.193945e-001 9.756262e-001 2.291839e-002
8.692506e-001 -1.193945e-001 2.890506e-001 1.838163e-001 -5.046334e-002
2.611794e-001 -9.756262e-001 -1.838163e-001 1.064105e+000 -6.831066e-001
-5.630523e-001 2.291839e-002 -5.046334e-002 6.831066e-001 2.752895e-002
```

13.2.5 半对数数据相关

1. 功能

对于给定的 n 个数据点 $(x_i,y_i)(i=0,1,\cdots,n-1)$ 用

$$y=bt^{ax},t>0$$

作拟合。

2. 算法

设给定 n 个数据点 $(x_i,y_i)(i=0,1,\cdots,n-1)$，且 $y_i>0$，用函数

$$y=bt^{ax},t>0$$

进行拟合。为了求拟合参数，两边取对数

$$\log_t y=\log_t b+ax$$

令

$$\tilde{y}=\tilde{a}\tilde{x}+\tilde{b}$$

其中

$$\tilde{y}=\log_t y,\tilde{x}=x,\tilde{a}=a,\tilde{b}=\log_t b$$

此时，问题化为对 n 个数据点 $(\tilde{x}_i,\tilde{y}_i)$ 作线性拟合，求出 $\tilde{a}$ 与 $\tilde{b}$ 后，就可以得到

$$a=\tilde{a},b=t^{\tilde{b}}$$

关于一元线性拟合可参看本章 13.2.2 节的算法说明。

3. 函数原型

```
//半对数数据相关
template <class _Ty>
void HalfLogarithmCorrelation(valarray<_Ty>& x, valarray<_Ty>& y, _Ty t, valarray<_Ty>& a);
```

4. 参数

x,y：均为 valarray<_Ty>类型引用，长度为 n。存放 n 个数据点 $(x_i,y_i)(i=0,1,\cdots,n-1)$，要求所有的 $y_i>0$。

t：_Ty 类型变量。指数函数的底，要求 $t>0$。

a:valarray<_Ty>类型引用,长度为 7。返回拟合函数的参数以及各种统计量,其意义如下。

a(0):拟合函数 $y=bt^{ax}$ 中的 b。

a(0):拟合函数 $y=bt^{ax}$ 中的 a。

a(2):偏差平方和 q,即 $q=\sum_{i=0}^{n-1}(y_i-bt^{axi})^2$。

a(3):平均标准偏差 s,即 $s=\sqrt{q/n}$。

a(4):最大偏差 umax,即 $umax=\max_{0\leqslant i\leqslant n-1}|y_i-bt^{axi}|$。

a(5):最小偏差 umin,即 $umin=\min_{0\leqslant i\leqslant n-1}|y_i-bt^{axi}|$。

a(6):偏差平均值 u,即 $u=\sum_{i=0}^{n-1}|y_i-bt^{axi}|/n$。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数中设数据点数为 n。

(2) 输入、输出的数据应为浮点数类型。

7. 示例

给定如下 12 个数据点:

x	0.96	0.94	0.92	0.90	0.88	0.86	0.84	0.82	0.80	0.78	0.76	0.74
y	558.0	313.0	174.0	97.0	55.8	31.3	17.4	9.70	5.58	3.13	1.74	1.00

用函数 $y=b10^{ax}$ 进行拟合,并求偏差平方和 q、平均标准偏差 s、最大偏差 umax、最小偏差 umin、偏差平均值 u。

主函数示例程序如下:

```
//HalfLogarithmCorrelation.cpp 半对数数据相关

#include <iostream>          //模板类输入、输出流 iostream 标准头文件
#include <valarray>          //模板类 valarray 的标准头文件
#include <Statistic.h>       //数据处理与回归分析头文件
using namespace std;         //名字空间

void main(void)
{
  long double t(10);
  valarray<long double> a(7);

  long double x[12] =
  {
      0.96,0.94,0.92,0.90,0.88,0.86,
      0.84,0.82,0.80,0.78,0.76,0.74
  };
```

```
    long double y[12] =
    {
        558.0,313.0,174.0,97.0,55.8,31.3,
        17.4, 9.70, 5.58, 3.13,1.74,1.00
    };

    valarray<long double> xx(x, 12);
    valarray<long double> yy(y, 12);

     HalfLogarithmCorrelation(xx, yy, t, a);

    printf("\n");
     printf("a = %13.7e b = %13.7e \n",a[1],a[0]);
     printf("\n");
     printf("q = %13.7e s = %13.7e \n",a[2],a[3]);
     printf("\n");
     printf("umax = %13.7e umin = %13.7e u = %13.7e \n",
                          a[4],a[5],a[6]);
    printf("\n");
}
```

8. 示例结果

```
a = 1.2492699e + 001 b = 5.6193146e - 010

q = 3.2713713e + 001 s = 1.6511035e + 000

umax = 5.0649866e + 000 umin = 1.1194247e - 002 u = 8.6305997e - 001
```

13.2.6 对数数据相关

1. 功能

对于给定的 n 个数据点 $(x_i,y_i)(i=0,1,\cdots,n-1)$ 用

$$y=bx^a$$

作拟合。

2. 算法

设给定 n 个数据点 $(x_i,y_i)(i=0,1,\cdots,n-1)$，且所有的 $x_i,y_i>0$，用函数

$$y=bx^a,x,y>0$$

进行拟合。

为了求拟合参数 a 与 b，两边取对数，即

$$\ln y=\ln b+a\ln x$$

令

$$\tilde{y}=\tilde{a}\tilde{x}+\tilde{b}$$

其中

$$\tilde{y}=\ln y,\tilde{x}=\ln x,\tilde{a}=a,\tilde{b}=\ln b$$

此时，问题化为对 n 个数据点 $(\tilde{x}_i,\tilde{y}_i)$ 作线性拟合，求出 $\tilde{a}$ 与 $\tilde{b}$ 后，就可以得到

$$a=\tilde{a},b=e^{\tilde{b}}$$

关于一元线性拟合参看本章 13.2.2 节的算法说明。

3. 函数原型

```
//对数数据相关
template<class_Ty>
Void LogarithmCorrelation (valarray<_Ty>&x,valarray<_Ty>&y,valarray<_Ty>&a);
```

4. 参数

x,y：均为 valarray<_Ty>类型引用，长度为 n。存放 n 个数据点 $(x_i,y_i)(i=0,1,\cdots,n-1)$，要求所有的 $x_iy_i>0$。

a：valarray<_Ty>类型引用，长度为 7。返回拟合参数以及各种统计量，其意义如下。

a(0)：拟合函数 $y=bx^a$ 中的 b。

a(1)：拟合函数 $y=bx^a$ 中的 a。

a(2)：偏差平方和 q，即 $q=\sum_{i=0}^{n-1}(y_i-bx_i^a)^2$。

a(3)：平均标准偏差 s，即 $s=\sqrt{q/n}$。

a(4)：最大偏差 umax，即 $umax=\max\limits_{0\leqslant i\leqslant n-1}|y_i-bx_i^a|$。

a(5)：最小偏差 umin，即 $umin=\min\limits_{0\leqslant i\leqslant n-1}|y_i-bx_i^a|$。

a(6)：偏差平均值 u，即 $u=\sum_{i=0}^{n-1}|y_i-bx_i^a|/n$。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数中设数据点数为 n。

(2) 输入、输出的数据应为浮点数类型。

7. 示例

给定如下 10 个数据点：

x	0.1	1.0	3.0	5.0	8.0	10.0	20.0	50.0	80.0	100.0
y	0.1	0.9	2.5	4.0	6.3	7.8	14.8	36.0	54.0	67.0

用函数 $y=bx^a$ 进行拟合，并求偏差平方和 q、平均标准偏差 s、最大偏差 umax、最小偏差 umin、偏差平均值 u。

主函数示例程序如下：

```
//LogarithmCorrelation.cpp  对数数据相关

#include <iostream>          //模板类输入输出流 iostream 标准头文件
#include <valarray>          //模板类 valarray 的标准头文件
#include <Statistic.h>       //统计 Statistic 头文件
using namespace std;         //名字空间

void main(void)
```

```
{
  valarray<long double> a(7);

  long double x[10] = {0.1,1.0,3.0,5.0,8.0,10.0,20.0,50.0,80.0,100.0};
  long double y[10] = {0.1,0.9,2.5,4.0,6.3,7.8,14.8,36.0,54.0,67.0};

  valarray<long double> xx(x, 10);
  valarray<long double> yy(y, 10);
  LogarithmCorrelation(xx, yy, a);

  printf("\n");
  printf("a = %13.7e b = %13.7e \n",a[1],a[0]);
  printf("\n");
  printf("q = %13.7e s = %13.7e \n",a[2],a[3]);
  printf("\n");
  printf("umax = %13.7e umin = %13.7e u = %13.7e \n",
                        a[4],a[5],a[6]);
  printf("\n");
}
```

8. 示例结果

```
a = 9.4157567e - 001 b = 8.8577259e - 001

q = 1.7865958e + 000 s = 4.2268142e - 001

umax = 8.5616856e - 001 umin = 1.3319881e - 003 u = 2.5061534e - 001
```

第14章 极 值

本章介绍计算极值的算法，每种算法都用C++语言以函数的形式实现。14.1节介绍极值的相关基本概念，以及极值算法程序原型的头文件，14.2节给出了计算极值算法的使用示例及相关说明。

14.1 极值与头文件

本节介绍极值的相关基本概念，并包括极值算法与程序原型，其算法与程序原型包含在头文件 Extremum.h 中，函数的定义内容在 Extremum.inl 文件中。

本章介绍的极值算法及程序包括：

- 一维极值连分式法；
- n 维极值连分式法；
- 线性规划；
- n 维极值单形调优法；
- n 维极值复形调优法；
- 确定一元函数极小值点所在区间；
- 一元函数极小值的黄金分割法；
- 一元函数极小值的不用导数布伦特法。

14.1.1 基本概念

求函数极值的数值方法或试验最优化方法有时称为“数学规划”。数学规划除了线性规划外统称为非线性规划。数学规划所处理的一般是静态问题，变分法、最大(小)值原理和动态规划所处理的一般是动态问题，但两者并无截然的界限。

14.1.2 Extremum.h

```
// Extremum.h            计算极值函数头文件
// Ver 1.0.0.0
// 版权所有(C) 何渝，2002
// 最后修改：2002.9.6.

#ifndef _EXTREMUM_H     //避免多次编译
#define _EXTREMUM_H

#include <comm.h>       //公共头文件
#include <random.h>     //随机数头文件
#inclnde <Matrix.h>     //矩阵类及相关函数头文件
```

```
#include <valarray>   //模板类 valarray 的标准头文件
using namespace std;      //名字空间

//template <class _Ty = float>
//一维极值连分式法
template <class _Ty>
void ExtremumFraction1D(valarray<_Ty>& x, _Ty eps,int k, valarray<int>& js);

//n 维极值连分式法
template <class _Ty>
void ExtremumFractionND(valarray<_Ty>& x, _Ty eps, int k, valarray<int>& js);

//线性规划
template <class _Ty>
int ExtremumLinePrograming(matrix<_Ty>& a, valarray<_Ty>& b, valarray<_Ty>& c,
                                                          valarray<_Ty>& x);

//n 维极值单形调优法
template <class _Ty>
int ExtremumSimplexND(_Ty d, _Ty u, _Ty v, valarray<_Ty>& x,
            _Ty eps, int k, matrix<_Ty>& xx, valarray<_Ty>& f);

//n 维极值复形调优法
template <class _Ty>
int ExtremumComplexND(int m, valarray<_Ty>& a, valarray<_Ty>& b,
     _Ty alpha, _Ty eps, valarray<_Ty>& x, matrix<_Ty>& xx, int k);

//确定一元函数极小值点所在区间
template <class _Ty>
void MinimizerInterval(_Ty& ax, _Ty& bx, _Ty& cx, _Ty& fa, _Ty& fb, _Ty& fc);

//确定一元函数极小值点所在区间
template <class _Ty>
int MinimizerInterval(_Ty& ax, _Ty& bx, _Ty& cx, int& sign, _Ty b, _Ty& fa, _Ty& fb, _Ty& fc);
//一元函数极小值的黄金分割法
template <class _Ty>
Ty ExtremumGold1D(_Ty ax, _Ty bx, _Ty cx, int sign, _Ty tol, _Ty& xmin);

//一元函数极小值的不用导数布伦特(Brent)法
template <class _Ty>
_Ty ExtremumBrentNonDerivative1D(_Ty ax, _Ty bx, _Ty cx, int sign,
                                          _Ty tol, _Ty& xmin);

#include "Extremum.inl"     //类及相关函数的定义头文件
```

```
#endif  // _EXTREMUM_H
```

14.2 极值数值计算方法

14.2.1 一维极值连分式法

1. 功能

用连分式法求目标函数 $f(x)$的极值点。

2. 算法

设目标函数 $f(x)$在区间$[a,b]$内可导，且单峰(或单谷)。$f(x)$的极值点为函数

$$y(x)=\frac{\mathrm{d}[f(x)]}{\mathrm{d}x}$$

的零点。

设 $y(x)$的反函数为 $x=\varphi(y)$，且用如下连分式表示

$$x=\varphi(y)=b_0+\frac{y-y_0}{b_1+}\frac{y-y_1}{b_2+}\cdots+\frac{y-y_{i-1}}{b_i+\cdots}$$

令 $y=0$，即得 $y(x)$的零点(即 $y(x)$的极值点)为

$$\tilde{x}=b_0-\frac{y_0}{b_1-}\frac{y_1}{b_2-}-\cdots-\frac{y_{i-1}}{b_i-\cdots}$$

由此可得计算 $f(x)$的极值点(即 $y(x)$的极值点)的过程如下。

在区间$[a,b]$内选取 3 个初始点 x_0，x_1 和 x_2，并分别计算相应的导数值

$$y_0=f'(x_0),y_1=f'(x_1),y_2=f'(x_2)$$

再利用连分式插值法计算系数 b_0，b_1 和 b_2。如此，又可以求得一个新的点

$$x_3=b_0-\frac{y_0}{b_1}-\frac{y_1}{b_2}$$

一般来说，假设已经取得点列$(x_0,y_0),(x_1,y_1),\cdots,(x_{i-1},y_{i-1})$，并由此确定了系数 b_0，$b_1,\cdots b_{i-1}$，则可求出新的点

$$x_i=b_0-\frac{y_0}{b_1}-\frac{y_1}{b_2}-\cdots\frac{y_{i-2}}{b_{i-1}}$$

及

$$y_i=f'(x_i)$$

再由如下递推关系计算出新的 b_i

$$b_0=x_0$$

$$b_{0i}=x_i$$

$$b_{j+1,i}=(y_i-y_j)/(b_{ji}-b_j),j=0,1,\cdots,i-1$$

$$b_i=b_{ii}$$

以上过程一直进行到对于某个 x_i 满足

$$|y_i|<\varepsilon$$

为止。此时 x_1 即为所求的极值 $\tilde{x}$。

求出极值点后，取一个很小的 Δx，若不等式

$$f(\tilde{x}+\Delta x)-2f(\tilde{x})+f(\tilde{x}-\Delta x)\leqslant 0$$

成立，则点 $\tilde{x}$ 为极大值点；反之为极小值点。

3. 函数原型

```
//一维极值连分式法
template <class _Ty>
void ExtremumFraction1D(valarray<_Ty>& x, _Ty eps,
int k, valarray<int>& js);
```

4. 参数

x：valarray<_Ty>类型引用，长度为 2。其中 $x(0)$存放极值点初值，返回时存放极值点 $\tilde{x}$；$x(1)$返回极值点 $\tilde{x}$ 处的函数值。

eps：_Ty 类型变量。控制精度要求。一般取 $10^{-10}\sim10^{-35}$ 之间的数。

k：整型变量。允许迭代次数的最大值。

js：valarray<int>类型引用，长度为 2，其中 $js(0)$返回实际迭代次数；$js(1)$返回标志。若 $js(0)=k$，则有可能没有满足精度要求，返回的极值点只作为参考。若 $js(1)<0$，则说明返回的极值点为极小值点；若 $js(1)>0$，则说明返回的极值点为极大值点。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数要调用计算目标函数 $f(x)$值与导数 $f'(x)$值的函数 FunctionValueEF1D()，由用户自己编写，其形式为：

```
template <class _Ty>
void FunctionValueEF1D(_Ty x, valarray<_Ty>& y)
{
   y[0] = f(x) 的表达式;
   y[1] = f'(x) 的表达式;
}
```

(2) 除参数 js 外，输入、输出的数据应为浮点数类型。

7. 示例

用连分式法计算目标函数

$$f(x)=(x-1)(10-x)$$

的极值点与极值点处的函数值。取初值 $x(0)=1.0$，$eps=10^{-30}$，最大迭代次数 $k=10$。

主函数及计算 $f(x)$和 $f'(x)$函数值的示例程序如下：

```
//ExtremumFraction1D.cpp 一维极值连分式法

#include <iostream>     //模板类 iostream 输入、输出流标准头文件
#include "Extremum.h"   //极值头文件

void main(void)
{
```

```
valarray<int> js(2);
valarray<long double> x(2);
long double eps = LONGDOUBLEERROR;

x[0] = 1;

ExtremumFraction1D(x, eps, 10, js);

cout << "js[0] = " << js[0] << endl;
if (js[1] < 0)
    cout << "MIN: ";
 else
   cout << "MAX: ";
cout << "x = " << x[0] << "\t" << "f(x) = " << x[1] << endl;
}

//*  计算函数及导数值
template <class _Ty>
void FunctionValueEF1D(_Ty x, valarray<_Ty>& y)
{
 y[0] = (x - 1.0) * (10.0 - x);   //计算 f(x)函数值
   y[1] = -2.0 * x + 11.0;     //计算 f(x)导数值
}
```

8. 示例结果

```
t = 0.15   z = 0.860709
t = 0.55   z = 0.57695
```

14.2.2 n 维极值连分式法

1. 功能

用连分式法计算多元函数的极值点与极值点处的函数值。

2. 算法

设 n 元函数

$$Z = f(x_0, x_1, \cdots, x_{n-1})$$

单峰(或单谷),则 $f(x_0, x_1, \cdots, x_{n-1})$的极值为方程组

$$\frac{\partial f}{\partial x_i} = 0, i = 0, 1, \cdots, n-1$$

的一组实数解。

利用连分式法,轮流求某个方向 x_i 上的极值点(其余 $n-1$ 个变量保持当前值不变)。即对于 $i=0,1,\cdots,n-1$,分别求函数

$$y(x_i) = \frac{\partial f}{\partial x_i}$$

的零点(具体方法见本章 14.2.1 节的算法说明)。

反复进行上述过程,直到满足

$$\sum_{i=0}^{n-1}\left|\frac{\partial f}{\partial x_i}\right|<\varepsilon$$

为止。

3. 函数原型

```
//n维极值连分式法
template <class _Ty>
void ExtremumFractionND(valarray<_Ty>& x, _Ty eps, int k, valarray<int>& js);
```

4. 参数

x:valarray<_Ty>类型引用,长度为 $n+1$。其中 $x(0),x(1),\cdots,x(n-1)$ 存放极初值,返回时存放极值点的 n 个坐标;$x(n)$返回极值点处的函数值。

eps:_Ty 类型变量,控制精度要求。

k: int 类型变量,允许最大迭代次数。

js: valarray<int>类型引用,长度为 2,其中 $js(0)$返回实际迭代次数;$js(1)$返回标志。若 $js(0)=k$,则有可能未满足精度要求,返回的极值点只作为参考。若 $js(1)<0$,则说明返回的为极小值点;若 $js(1)>0$,则说明返回的为极大值点。$js(1)=0$,则说明返回的为鞍点。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数中自变量个数为 n。

(2) 本函数要调用计算目标函数 $f(x_0,x_1,\cdots,x_{n-1})$ 以及各偏导数值的函数 FunctionValueEFND(),由用户自己编写,其形式为:

```
template <class _Ty>
_Ty FunctionValueEFND(valarray<_Ty>& x, int j)
{
    _Ty y;
    int n = x.size() - 1;                    //自变量个数
    switch(j)
    {
      case 0:y = f(x0,x1,…,xn-1)的表达式;break;
      case 1: y = ∂f/∂x0的表达式;break;
                  ⋮
      case n: y = ∂f/∂xn-1的表达式;break;
      default: { }
    }
    return(y);
}
```

(3) 除参数 js 外,输入、输出的数据应为浮点数类型。

7. 示例

用连分式法求二元函数

$$z=(x_0-1)^2+(x_1+2)^2+2$$

的极值点与极值点处的函数值。

取初值 $x_0=0.0, x_1=0.0$ ，$eps=10^{-30}$，最大迭代次数 $k=10$。

主函数及计算各偏导数值的函数示例程序如下：

```
//Extremum FractionND.cpp n维极值连分式法

#include <iostream>              //模板类 iostream 输入、输出流标准头文件
#include "Extremum.h"            //极值头文件
void main(void)
{
 valarray<int> js(2);
 long double eps = LONGDOUBLEERROR;
 valarray<long double> x(3);

 x[0] = x[1] = 0;

   ExtremumFractionND(x, eps, 10, js);

 cout << "js[0] = " << js[0] << endl;
 cout << " x[0] = " << x[0] << endl;
 cout << " x[1] = " << x[1] << endl;

 if (js[1] < 0)
     cout << "MIN: ";
   else
     cout << "MAX: ";

 cout << "z = " << x[2] << endl;
}

// * 计算函数及导数值
template <class _Ty>
_Ty FunctionValueEFND(valarray<_Ty>& x, int j)
{
  _Ty y;

  int n = x.size() - 1;          //自变量个数

  switch(j)                      //计算 f(x)函数值和导数值
  {
     case 0: y = (x[0] - 1.0) * (x[0] - 10.0) + (x[1] + 2.0) * (x[1] + 2.0) + 2.0;
             break;
     case 1: y = 2.0 * (x[0] - 1.0);
             break;
     case 2: y = 2.0 * (x[1] + 2.0);
```

```
            break;
    }
      return(y);
    }
```

8. 示例结果

```
 js[0] = 2
  x[0] = 1
  x[1] = -2
 MIN: z = 2
```

14.2.3 线性规划

1. 功能

求解不等式约束条件下的线性规划问题。

2. 算法

设给定 m 阶 n 维不等式约束条件

$$\begin{cases} a_{00}x_0 + a_{01}x_1 + \cdots + a_{0,n-1}x_{n-1} \leqslant b_0 \\ a_{10}x_0 + a_{11}x_1 + \cdots + a_{1,n-1}x_{n-1} \leqslant b_1 \\ \quad \vdots \\ a_{m-1,0}x_0 + a_{m-1,1}x_1 + \cdots + a_{m-1,n-1}x_{n-1} \leqslant b_{m-1} \end{cases}$$

且 $x_j \geqslant 0, j=0,1,\cdots,n-1$

求一组$(x_0, x_1, \cdots, x_{n-1})$值，使目标函数

$$f = \sum_{j=0}^{n-1} c_j x_j$$

达到极小值。

如果求极大值，则只要令 $\tilde{f} = -f$，此时就化为求目标函数

$$\tilde{f} = -\sum_{j=0}^{n-1} c_j x_j$$

的极小值。

引进 m 个非负松弛变量 $x_n, x_{n+1}, \cdots, x_{n+m-1}$，则上述问题化为：

寻找 X 值，使满足

$$AX = B \tag{1}$$

$$X \geqslant 0 \tag{2}$$

且使目标函数

$$f = C^{\mathrm{T}}X \tag{3}$$

达到极小值。其中

$$X = (x_0, x_1, \cdots, x_{n-1}, x_n, \cdots, x_{n+m-1})^{\mathrm{T}}$$

$$B = (b_0, b_1, \cdots, b_{m-1})^{\mathrm{T}}$$

$$C = (C_0, C_1, \cdots, C_{n-1}, 0, \cdots, 0)^{\mathrm{T}}$$

$$\boldsymbol{A}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} & 1 & 0 & \cdots & 0 \\ a_{10} & a_{11} & \cdots & a_{1,n-1} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} & 0 & 0 & \cdots & 1 \end{bmatrix}$$

称满足(1)和(2)的解为容许解，其中正分量的个数不多于 m 个的容许解称为基本解，而使(3)取极小值的解称为最优解，最优解必在基本解中。

寻找最优解的过程如下：

假定已得到一个基本解，其正分理个数为 m，设分别为 $x_{i_0},x_{i_1},\cdots,x_{i_{m-1}}$，且与之对应的矩阵 $\boldsymbol{A}$ 中的列向量 $P_{i_0},P_{i_1},\cdots,P_{i_{m-1}}$ 为线性无关，则这组向量称为基底向量。对于矩阵 $\boldsymbol{A}$ 中的每一列向量均可用基底向量的线性组合表示，即

$$\boldsymbol{A}=\boldsymbol{PD}$$

其中

$$\boldsymbol{P}=[P_{i_0},P_{i_1},\cdots,P_{i_{m-1}}]$$

$$\boldsymbol{D}=\begin{bmatrix} d_{00} & d_{01} & \cdots & d_{0,n+m-1} \\ d_{10} & d_{11} & \cdots & d_{1,n+m-1} \\ \vdots & \vdots & & \vdots \\ d_{m-1,0} & d_{m-1,1} & \cdots & d_{m-1,n+m-1} \end{bmatrix}$$

而组合系数矩阵 $\boldsymbol{D}$ 可用下式计算

$$\boldsymbol{D}=\boldsymbol{P}^{-1}\boldsymbol{A}$$

令 $z_j=c_{i_0}d_{0j}+c_{i_1}d_{1j}+\cdots+c_{i_{m-1}}d_{m-1,j}$，$j=0,1,\cdots,n+m-1$

如果对于所有的 $j(j=0,1,\cdots,n+m-1)$ 满足

$$z_j-c_j\leqslant 0$$

则取 X 为

$$(x_{i_0},x_{i_1},\cdots,x_{i_{m-1}})^{\mathrm{T}}$$

而 X 的其余 n 个分量均为 0，此时 X 即为最优解。

否则，选择对应于

$$\max_{0\leqslant j\leqslant n+m-1}(z_j-c_j)=z_k-c_k$$

的向量 P_k 进入基底向量组，而将对应于

$$\min_{0\leqslant l\leqslant m-1}(x_{i_l}/d_{lk})=x_{i_l}/d_{lk}$$

$$d_{lk}>0$$

的向量 P_l 从基底向量组中消去。这样，对应新的基底，其目标函数值比原先的下降了。如果 $d_{lk}\leqslant 0(l=0,1,\cdots,m-1)$，则说明目标函数无界。

重复以上过程，直至求出最优解，或者确定目标函数值无界为止。

在上述的每一步中，新的解可用下式计算

$$(x_{i_0},x_{i_1},\cdots,x_{i_{m-1}})^{\mathrm{T}}=\boldsymbol{P}^{-1}\boldsymbol{B}$$

基底向量组的初值取单位矩阵，即

$$\boldsymbol{P}=\boldsymbol{I}_m$$

也就是说，取初始解为

$$\boldsymbol{X}=(0,0,\cdots,0,b_0,b_1,\cdots,b_{m-1})^{\mathrm{T}}$$

3. 函数原型

```
//线性规划
template <class _Ty>
int ExtremumLinePrograming(matrix<_Ty>& a, valarray<_Ty>& b,
                           valarray<_Ty>& c, valarray<_Ty>& x)
```

4. 参数

a:matrix<_Ty>类型引用体积为 $m\times(m+n)$。存放不等式的约束条件中左端系数矩阵及 m 阶单位矩阵,其存放格式为

$$\boldsymbol{A}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} & 1 & 0 & \cdots & 0 \\ a_{10} & a_{11} & \cdots & a_{1,n-1} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} & 0 & 0 & \cdots & 1 \end{bmatrix}$$

b:valarray<_Ty>类型引用,长度 m,存放不等式约束条件的右端项 $b_0,b_1,\cdots,b_{m-1}$。

c:valarray<_Ty>类型引用,长度 $m+n$,存放目标函数中的系数,其中后 m 个分量为 0,即存放顺序为

$$c=(c_0,c_1,\cdots,c_{n-1},0,\cdots,0)$$

m:整型变量。不等式约束条件的个数。

n:整型变量。变量个数。

x:valarray<_Ty>类型引用,长度为 $m+n$。其中前 n 个分量返回目标函数 f 的极小值点的 n 个坐标(即最优解),第 $n+1$ 个分量(即 $x(n)$)返回目标函数 f 的极小值。其余为本函数的工作单元。

5. 返回值

int 类型。

0:表示目标函数值无界。

非 0:表示正常返回。

6. 函数说明

(1) 本函数要调用全选主元高斯—约当法求逆的函数 MatrixInversionGS(),参看 5.5.1 节。

(2) 本函数用到实矩阵相乘的运算,请参看 5.2.3 节。

(3) 除返回值外,输入、输出的数据应为浮点数类型。

(4) 本函数中不等式约束条件的个数为 m,变量的个数为 n。

7. 示例

设不等式约束条件为

$$\begin{cases} x_0+2x_1+7x_2\leqslant 10 \\ x_0+4x_1+13x_2\leqslant 18 \\ 2x_1+8x_2\leqslant 13 \\ x_0,x_1,x_2\geqslant 0 \end{cases}$$

求目标函数 $f=4x_0+9x_1+26x_2$ 的极大值。

化为极小值问题后的目标函数为 $\tilde{f}=-4x_0-9x_1-26x_2$。

在本例中，$m=3, n=3, m+n=6$ 且

$$\boldsymbol{A}=\begin{bmatrix}1 & 2 & 7 & 1 & 0 & 0\\ 1 & 4 & 13 & 0 & 1 & 0\\ 0 & 2 & 8 & 0 & 0 & 1\end{bmatrix}$$

$$\boldsymbol{B}=(10,18,13)^{\mathrm{T}}$$

$$\boldsymbol{C}=(-4,-9,-26,0,0,0)^{\mathrm{T}}$$

$$\boldsymbol{X}=(x_0,x_1,x_2,x_3,x_4,x_5)^{\mathrm{T}}$$

主函数及示例程序如下：

```
//ExtremumLinePrograming.cpp 线性规划

#include <iostream>                    //模板类 iostream 输入、输出流标准头文件
#include "Extremum.h"                  //极值头文件

void main(void)
{
 valarray<long double> x(6);
 long double aa[3][6] = {    {1, 2, 7, 1, 0, 0},
                             {1, 4, 13, 0, 1, 0},
                             {0, 2, 8, 0, 0, 1}};
 long double bb[3] = {10, 18, 13};
 long double cc[6] = {-4, -9, -26, 0, 0, 0};

 matrix<long double> a(&aa[0][0], 3, 6);
 valarray<long double> b(bb, 3);
 valarray<long double> c(cc, 6);

 int iValue = ExtremumLinePrograming(a,b,c,x);

 if(iValue ! = 0)
 {
     for (iValue = 0; iValue < 6; iValue + +)
         cout << "x(" << iValue << ") = " << x[iValue] << endl;
 }
}
```

8. 示例结果

```
x(0) = 2
x(1) = 4
x(2) = 0
x(3) = -44
x(4) = 0
x(5) = 5
```

14.2.4 n 维极值单形调优法

1. 功能

用单形调优法求解无约束条件下的 n 维极值问题。

2. 算法

设具有 n 个变量的目标函数为 $J=f(x_0,x_1,\cdots,x_{n-1})$。

单形调优法求目标函数 J 的迭代过程如下。

(1) 在 n 维变量空间中确定一个由 $n+1$ 个顶点所构成的初始单形

$$X_{(i)}=(x_{0i},x_{1i},\cdots,x_{n-1,i}),i=0,1,\cdots,n$$

并计算在每一个顶点上的函数值

$$f_{(i)}=f(X_{(i)}),i=0,1,\cdots,n$$

(2) 确定

$$f_{(R)}=f(X_{(R)})=\max_{0\leqslant i\leqslant n}f_{(i)}$$

$$f_{(G)}=f(X_{(G)})=\max_{\substack{0\leqslant i\leqslant n\\ i\neq R}}f_{(i)}$$

$$f_{(L)}=f(X_{(L)})=\min_{0\leqslant i\leqslant n}f_{(i)}$$

其中 $X_{(R)}$ 称为最坏点。

(3) 求出最坏点 $X_{(R)}$ 的对称点 $X_T=2X_F-X_{(R)}$，其中 $X_F=\dfrac{1}{n}\sum\limits_{\substack{i=0\\ i\neq R}}^{n}X_{(i)}$。

(4) 确定新的顶点替代原顶点，从而构成新的单形。替代的原则如下：

若 $f(X_T)<f_{(L)}$，则需要由下式将 X_T 扩大为 X_E：

$$X_E=(1+\mu)X_T-\mu X_F$$

其中 μ 称为扩张系数，一般 $1.2<\mu<2.0$，在这种情况下，如果 $f(X_E)=f_{(L)}$，则 $X_E\Rightarrow X_{(R)}$，$f(X_E)\Rightarrow f_{(R)}$；否则 $X_T\Rightarrow X_{(R)}$，$f(X_T)\Rightarrow f_{(R)}$。

若 $f(X_T)\leqslant f_{(G)}$，则 $X_T\Rightarrow X_{(R)}$，$f(X_T)\Rightarrow f_{(R)}$。

若 $f(X_T)\leqslant f_{(G)}$，如果 $f(X_T)>f_{(R)}$，则 $X_T\Rightarrow X_{(R)}$，$f(X_T)\Rightarrow f_{(R)}$。然后由下式将 X_T 缩小为 X_E：

$$X_E=\lambda X_{(R)}+(1-\lambda)X_E$$

其中 λ 称为收缩系数，一般取 $0.0<\lambda<1.0$。在这种情况下，如果

$$f(X_E)\Rightarrow X_{(R)}$$

则新的单形的 $n+1$ 个顶点为

$$X_{(i)}=\frac{1}{2}(X_{(i)}+X_{(L)}),i=0,1,\cdots,n$$

且计算 $f_{(i)}=f(X_{(i)}),i=0,1,\cdots,n$；否则，$X_E\Rightarrow X_{(R)}$，$f(X_E)\Rightarrow f_{(R)}$

重复(2)～(4)，直到单形中各顶点距离小于预先给定的精度要求为止。

如果实际问题中需要求极大值，则只要令目标函数为

$$\tilde{J}=-J=-f(x_0,x_1,\cdots,x_{n-1})$$

即可，此时 $\tilde{J}$ 的极小值的绝对值即为 J 的极大值。

3. 函数原型

```
//n维极值单形调优法
```

```
template <class _Ty>
int ExtremumSimplexND(_Ty d, _Ty u, _Ty v, valarray<_Ty>& x,
                      _Ty eps, int k, matrix<_Ty>& xx, valarray<_Ty>& f);
```

4. 参数

d:_Ty 类型变量。初始单形中任意两顶点间的距离。

u:_Ty 类型变量。存放扩张系数 μ,一般取 $1.2<\mu<2.0$。

v:_Ty 类型变量。存放扩张系数 λ,一般取 $0.0<\lambda<1.0$。

x:valarray<_Ty>类型引用,长度为 $n+1$。其中前 n 个分量返回极小值点的 n 个坐标,最后一个分量(即 $x(n)$)返回极小值。

eps:_Ty 类型变量。控制精度要求。

k:整型变量。允许最大迭代次数。

xx:matrix<_Ty>类型引用,体积为 $n\times(n+1)$。返回最后单形的 $n+1$ 个顶点的坐标。

$$(x_{0i},x_{1i},\cdots,x_{n-1,i}),i=0,1,\cdots,n$$

f:valarray<_Ty>类型引用,长度为 $n+1$。返回最后单形的 $n+1$ 个顶点上的目标函数值。

5. 返回值

int 类型。

本函数返回实际迭代数。若实际迭代数等于允许的最大迭代次数 k,则有可能未达到精度要求,返回的极值点只作参考。

6. 函数说明

(1) 本函数要调用计算目标函数 $f(x_0,x_1,\cdots,x_{n-1})$的值的函数 Function Value ESND(),由用户自己编写,其形式为:

```
template <class _Ty>
_Ty FunctionValueESND(valarray<_Ty>& x, int n)
{
   _Ty y;

   y = f(x0,x1,…,xn-1)的表达式;

   return(y);
}
```

(2) 本函数变量的个数为 n。

(3) 除返回值外,输入、输出的数据应为浮点数类型。

7. 示例

用单形调优法求目标函数

$$J=100(x_1-x_0^2)+(1-x_0)^2$$

的极小值点与极小值。

取 $eps=10^{-30}$,$d=1.0$,$\mu=1.6$,$\lambda=0.4$,$k=200$。

主函数及及计算目标函数值函数的示例程序如下:

```
//Extremum SimplexND.cpp n维极值单形调优法

#include <iostream>          //模板类 iostream 输入、输出流标准头文件
```

```
#include "Extremum.h"          //极值头文件
void main(void)
{
 valarray<long double> x(3);
 valarray<long double> f(3);
 matrix<long double> xx(2,3);
 long double eps = LONGDOUBLEERROR, d = 1.0, u = 1.6, v = 0.4;

int i = ExtremumSimplexND(d,u,v,x,eps,200,xx,f);

cout << "i = " << i << endl << endl;

for (i = 0; i<3; i++)
   cout << "x(0) = " << xx(0,i)<< "\t x(1) = " << xx(1,i)
                                  << "\t f = " << f[i] << endl;
cout << endl;

  for(i = 0; i<2; i++)
    cout << "x(" << i << ") = " << x[i] << endl;
cout << endl;

  cout << "J = " << x[2] << endl;
}

//计算 f(X)函数值
template <class _Ty>
_Ty FunctionValueESND(valarray<_Ty>& x, int n)
{
_Ty y;

  y = x[1] - x[0] * x[0];
y = 100.0 * y * y;
  y = y + (1.0 - x[0]) * (1.0 - x[0]);

  return(y);
}
```

8. 示例结果

```
i = 87

x(0) = 1   x(1) = 1   f = 2.44509e-016
x(0) = 1   x(1) = 1   f = 3.84618e-016
x(0) = 1   x(1) = 1   f = 1.93161e-016

x(0) = 1
x(1) = 1
```

J = 4.09569e-017

14.2.5 n 维极值复形调优法

1. 功能

用复型调优法解等式与不等式约束条件下的 n 维极值问题。

2. 算法

设多变量目标函数为 $J=f(x_0,x_1,\cdots,x_{n-1})$，$n$ 个常量约束条件为 $a_i\leqslant x_i\leqslant b_i$，$i=0,1,\cdots,n-1$，$m$ 个函数约束条件为

$$C_j(x_0,x_1,\cdots,x_{n-1})\leqslant W_j(x_0,x_1,\cdots,x_{n-1})\leqslant D_j(x_0,x_1,\cdots,x_{n-1})$$
$$j=0,1,\cdots,m-1$$

求 n 维目标函数 J 的极小值点及极小值。

若实际问题中要求极大值，只需令

$$\tilde{J}=-J=-f(x_0,x_1,\cdots,x_{n-1})$$

$\tilde{J}$ 的极小值的绝对值即为 J 的极大值。

复形调优法求目标函数 J 的极小值点的迭代过程如下：

复形共有 $2n$ 个顶点。假设给定初始复形中的第一个顶点坐标

$$X_{(0)}=(x_{00},x_{10},\cdots,x_{n-1,0})$$

且此顶点坐标满足 n 个常数约束条件及 m 个函数约束条件。

(1) 在 n 维变量空间中再确定出初始复形的其余 $2n-1$ 个顶点。其方法如下：

利用伪随机数按常量约束条件产生第 j 个顶点 $X_{(j)}=(x_{0j},x_{1j},\cdots,x_{n-1,j})$($j=1,2,\cdots,2n-1$)中的各分量 x_{ij}((($i=1,0,\cdots,n-1$)，即

$$x_{ij}=a_i+r(b_i-a_i),i=1,0,\cdots,n-1,j=1,2,\cdots,2n-1$$

其中 r 为[0,1]之间的一个伪随机数。

显然，由上述方法产生的初始复形的各顶点满足常量约束条件。然后再检查它们是否符合函数约束条件，如果不符合，则需要作调整，直到全部顶点均符合函数约束条件及常量约束条件为止。调整的原则为：

假设前 j 个顶点已满足所有的约束条件，而第 $j+1$ 个顶点不满足约束条件，则作如下调整变换($j=1,2,\cdots,2n-1$)：

$$X_{(j+1)}=(X_{(j+1)}+T)/2$$

其中

$$T=\frac{1}{j}\sum_{k=1}^{j}X_{(k)}$$

这个过程一直作到满足所有约束条件为止。

初始复形的 $2n$ 个顶点确定以后，计算各顶点处的目标函数值

$$f_{(j)}=f(X_{(j)}),j=0,1,\cdots,2n-1$$

(2) 确定

$$f_{(R)}=f(X_{(R)})=\max_{0\leqslant i\leqslant 2n-1}f_{(i)}$$
$$f_{(G)}=f(X_{(G)})=\max_{\substack{0\leqslant i\leqslant 2n-1\\ i\neq R}}f_{(i)}$$

其中 $X_{(R)}$ 称为最坏点。

(3) 计算最坏点 $X_{(R)}$ 的对称点

$$X_T = (1+\alpha)X_F - \alpha X_{(R)}$$

其中

$$X_F = \frac{1}{2n-1}\sum_{\substack{i=0 \\ i\neq R}}^{2n-1} X_{(i)}$$

α 称为反射系数，一般取 1.3 左右。

(4) 确定一个新的顶点替代最坏点 $X_{(R)}$ 以构成新的复形。其方法如下：

如果 $f(X_T)>f_{(G)}$，则用下式修改 X_T

$$X_T = (X_F + X_T)/2$$

直到 $f(X_T) \leqslant f_{(G)}$ 为止。

然后检查 X_T 是否满足所有约束条件。如果对于某个分量 $X_T(j)$ 不满足常量约束条件，即如果 $X_T(j)<a_j$ 或 $X_T(j)>b_j$，则令

$$X_T(j) = a_j + \delta \text{ 或 } X_T(j) = b_j - \delta$$

其中 δ 为很小的一个常数，一般取 $\delta=10^{-6}$。然后重复步骤(4)。

如果 X_T 不满足函数约束条件，则用下式修改 X_T

$$X_T = (X_F + X_T)/2$$

重复步骤(4)。

直到 $f(X_T) \leqslant f_{(G)}$ 且 X_T 满足所有约束条件为止。此时令

$$X_{(R)} = X_T, f_{(R)} = f(X_T)$$

重复步骤(2)～(4)，直到复形中各顶点距离小于预先给定的精度要求为止。

3. 函数原型

```
//n维极值复形调优法
template <class _Ty>
int ExtremumComplexND(int m, valarray<_Ty>& a, valarray<_Ty>& b,
    _Ty alpha, _Ty eps, valarray<_Ty>& x, matrix<_Ty>& xx, int k);
```

4. 参数

m：int 类型变量。约束条件的个数。

a：valarray<_Ty>类型引用，长度为 n。依次存放常量约束条件中变量 $x_i(i=0,1,\cdots,n-1)$ 的下界。

b：valarray<_Ty>类型引用，长度为 n。依次存放常量约束条件中变量 $x_i(i=0,1,\cdots,n-1)$ 的上界。

alpha：_Ty 类型变量。反射系数 α，一般取值为 1.3 左右。

eps：_Ty 类型变量。控制精度要求。

x：valarray<_Ty>类型引用，长度为 $n+1$。其中前 n 个分量存放初始复形的第一个顶点坐标值(要求满足所有的约束条件)，返回时存放极小值点各坐标值，最后一个分量 $x(n)$ 返回极小值。

xx：matrix<_Ty>类型引用，体积为 $(n+1)\times 2$。其中前 n 行返回最后复形的 $2n$ 个顶点坐标(一列为一个顶点)

$(x_{0i}, x_{1i}, \cdots, x_{n-1,i}), i = 0, 1, \cdots, 2n-1$

最后一行返回最后复形的 $2n$ 个顶点上的目标函数值。

k:int 类型变量。允许最大迭代次数。

5. 返回值

int 类型。

返回实际迭代数。若实际迭代次数等于 k,则返回的极小值点可能不满足精度要求,只作为参考。

6. 函数说明

(1) 本函数要调用计算目标函数值的函数 FunctionValueTarget(),由用户自己编写,其形式如下:

```
template <class _Ty>
_Ty FunctionValueTarget(valarray<_Ty>& x, int n)
{
_Ty y;

y = f(x0, x1, ···, xn-1)的表达式;

return(y);
}
```

(2) 本函数要调用计算函数约束条件中的下限、上限及条件值的函数 ConditionValue(),由用户自己编写,其形式如下:

```
template <class _Ty>
void ConditionValue(int n, int m, valarray<_Ty>& x,
valarray<_Ty>& c, valarray<_Ty>& d, valarray<_Ty>& w)
{
        c[0] = C0(x0, x1, ···, xn-1)的表达式;
             ⋮
        c[m-1] = Cm-1(x0, x1, ···, xn-1)的表达式;
        d[0] = D0(x0, x1, ···, xn-1)的表达式;
             ⋮
        d[m-1] = Dm-1(x0, x1, ···, xn-1)的表达式;
        w[0] = W0(x0, x1, ···, xn-1)的表达式;
             ⋮
        w[m-1] = Wm-1(x0, x1, ···, xn-1)的表达式;
}
```

(3) 除参数 m 和 k 外,其余参数类型应为浮点数类型。

7. 示例

用复形调优法求目标函数

$$J = f(x_0, x_1) = -\frac{[9-(x-3)^2]2x_1^3}{27\sqrt{3}}$$

满足约束条件

$$\begin{cases} x_0 \geqslant 0 \\ x_1 \geqslant 0 \\ 0 \leqslant x_1 \leqslant \dfrac{x_0}{\sqrt{3}} \\ 0 \leqslant x_0 + \sqrt{3}x_1 \leqslant 6 \end{cases}$$

的极小值点及极小值。

取 $eps=10^{-30}$，反射系数 $alpha=1.3$。量大迭代次数 $k=200$。

初始复形的第一个顶点为(0.0,0.0)。

其中常量约束条件的下界为 $a_0=0.0, a_1=0.0$，上界取 $b_0=10^{35}, b_1=10^{35}$；函数约束条件下的下限、上限及条件函数为

$$C_0(x_0,x_1)=0.0, C_1(x_0,x_1)=0.0$$

$$D_0(x_0,x_1)=x_0/\sqrt{3}, D_1(x_0,x_1)=6.0$$

$$W_0(x_0,x_1)=x_1, W_1(x_0,x_1)=x_0+\sqrt{3}x_1$$

主函数及计算目标函数值、计算函数约束条件中各值的函数的示例程序如下：

```
//Extremum ComplexND.cpp n维极值复形调优法

#include <iostream>              //模板类 iostream 输入、输出流标准头文件
#include "Extremum.h"            //极值头文件

void main(void)
{
 valarray<long double> x(3);
 valarray<long double> a(2);
 valarray<long double> b(2);
 matrix<long double> xx(3,4);
 long double eps = LONGDOUBLEERROR, alpha = 1.3;

 x[0] = x[1] = 0;
 a[0] = a[1] = 0;
 b[0] = b[1] = 1.0e+35;

 int i = ExtremumComplexND(2,a,b,alpha,eps,x,xx,200);

 cout << "i = " << i << endl << endl;

 for (i=0; i<4; i++)
    cout << "x(0) = " << xx(0,i) << "\t x(1) = " << xx(1,i)
                                 << "\t f = " << xx(2,i) << endl;

 cout << endl;

 for (i=0; i<2; i++)
```

```
    cout << "x(" << i << ") = " << x[i] << endl;

 cout << endl;

  cout << "J = " << x[2] << endl;
}

//* 计算目标函数值
template <class _Ty>
_Ty FunctionValueTarget(valarray<_Ty>& x, int n)
{
  _Ty y;

    y = -(9.0 - (x[0] - 3.0) * (x[0] - 3.0));
    y = y * x[1] * x[1] * x[1] / (27.0 * sqrt(3.0));

    return(y);
}

//* 计算函数约束条件的上、下限及条件值
template <class _Ty>
void ConditionValue(int n,int m,valarray<_Ty>& x, valarray<_Ty>& c,
                                valarray<_Ty>& d, valarray<_Ty>& w)
{

  c[0] = c[1] = 0.0;

   d[0] = x[0] / sqrt(3.0);
  d[1] = 6.0;

   w[0] = x[1];
  w[1] = x[0] + x[1] * sqrt(3.0);

  return;
}
```

8. 示例结果

```
i = 85

x(0) = 3   x(1) = 1.73205  f = -1
x(0) = 3   x(1) = 1.73205  f = -1
x(0) = 3   x(1) = 1.73205  f = -1
x(0) = 3   x(1) = 1.73205  f = -1

x(0) = 3
```

```
x(1) = 1.73205

J = -1
```

14.2.6 确定一元函数极小值点所在区间

1. 功能

利用黄金比率和二次插值算法，以尽可能少的计算量确定一元目标函数 $f(x)$ 的极小值点所在的区间，即保证一元函数 $f(x)$ 的极小值点在这个区间内。

当要确定一元函数 $f(x)$ 极大值点时，只需确定 $-f(x)$ 的极小值点即可实现。

2. 算法

(1) 算法一

确定极小值点所在区间。

任选区间初始点 a 和 b，然后按如下步骤计算。

① 确定下降方向。设求解 $f(x)$ 的极小值，则计算和 $f(b)$。

若 $f(a)>f(b)$，则下降方向为从 a 到 b，沿 $a\rightarrow b$ 的方向按黄金比率选取一点 c；

若 $f(a)\leqslant f(b)$，将点 a 和 b 位置进行交换，则此时下降方向仍是从 a 到 b，沿此方向按黄金比率选取一点 c，并计算 $f(c)$。

② 若 $f(b)<f(c)$，则 $[a,c]$ 即为所求区间，计算结束。

若 $f(b)\leqslant f(c)$，利用 $(a,f(a))$、$(b,f(b))$ 和 $(c,f(c))$ 进行二次插值，求其极小值点 u 及 $u\mathrm{lim}=b+1.61803399(c-b)$。

③ 若 u 在 b 和 c 之间，计算 $f(u)$，并作判断：

若 $f(u)\leqslant f(c)$，是 $[b,c]$ 即为所求，计算结束；

若 $f(u)>f(b)$，则 $[a,u]$ 即为所求，计算结束；

若上面两条件均不满足，则用黄金比率重新选点 $u=c+1.61803399(c-b)$，并计算 $f(u)$。

④ 当 u 在 c 和 $u\mathrm{lim}$ 之间时，若 $f(u)>f(c)$，则极小值点所在区间即为 $[b,u]$，计算结束；若 $f(u)\leqslant f(c)$，则去掉离对应的极小值点最远的点，此时显然为 a，将 c 和 u 看作新的 b 和 c，转向②。若 u 已不在 c 和 $u\mathrm{lim}$ 之间，则将此时的 b，c 和 $u\mathrm{lim}$ 看作新的一组 a，b，c，转向②。

(2) 算法二

算法二与算法一无本质区别，只是算法二不但可以用来确定一元函数 $f(x)$ 极小值点所在区间，也可用来确定一元函数 $f(x)$ 极大值点所在区间，它们不同的地方如下。

① 在算法二中，当确定下降方向时，若 $f(a)>f(b)$，计算的过程没有变化，与算法一相同；但当 $f(a)\leqslant f(b)$，则不将点 a 和 b 位置进行交换，而使用一个标志，将原所有计算函数 $f(x)$ 值的地方变为计算 $-f(x)$，即变成确定 $-f(x)$ 的极小值点区间的过程。

② 当使用算法二时，可以用来协助求解闭区间 $[a,b]$ 上多峰连续函数的多个极值(极小值和极大值)，a，b 为计算机精度范围内可接受的有限数值。由此在算法二中专门设置了一个参数作为闭区间的右边界点值，在确定目标函数 $f(x)$ 极值点的计算过程中，若其区间的边界点超过该点，算法将中止，程序将结束运算，即在计算过程中判断是否有 $u>b$ 的情况出现。

3. 函数原型

```
//确定一元函数极小值点所在区间 对应于算法一
template <class _Ty>
```

```
void MinimizerInterval(_Ty& ax, _Ty& bx, _Ty& cx, _Ty& fa, _Ty& fb, _Ty& fc);

//确定一元函数极小值点所在区间 对应于算法二
template <class _Ty>
int MinimizerInterval(_Ty& ax, _Ty& bx, _Ty& cx, int& sign,
                                    _Ty b, _Ty& fa, _Ty& fb, _Ty& fc);
```

4. 参数

算法一所对应 MinimizerInterval 的参数如下。

ax:_Ty 类型引用,开始时输入存放初始区间端点,调用后存放并输出极小值点所在区间的一个端点。

bx:_Ty 类型引用,开始时输入存放初始区间端点,调用后存放并输出极小值点的初始近似。

cx:_Ty 类型引用,存放并输出极小值点所在区间的另一个端点。

fa:_Ty 类型引用,存放并输出目标函数 $f(x)$在点 ax 处的函数值。

fb:_Ty 类型引用,存放并输出目标函数 $f(x)$在点 bx 处的函数值。

fc:_Ty 类型引用,存放并输出目标函数 $f(x)$在点 cx 处的函数值。

算法二所对应 MinimizerInterval 的参数如下。

ax:_Ty 类型引用,开始时输入存放初始区间端点,调用后存放并输出极小值点所在区间的一个端点。

bx:_Ty 类型引用,开始时输入存放初始区间端点,调用后存放并输出极小值点的初始近似。

cx:_Ty 类型引用,存放并输出极小值点所在区间的另一个端点。

sign:int 类型引用,求一元函数 $f(x)$极小值或极大值标志,当 sign 为 1 时,求目标函数的极小值;当 sign 为 -1 时,求目标函数的极大值。

b:_Ty 类型,输入目标函数 $f(x)$的定义区间右边界点值。

fa:_Ty 类型引用,存放并输出目标函数 $f(x)$在点 ax 处的函数值。

fb:_Ty 类型引用,存放并输出目标函数 $f(x)$在点 bx 处的函数值。

fc:_Ty 类型引用,存放并输出目标函数 $f(x)$在点 cx 处的函数值。

5. 返回值

算法一无返回值。

算法二的返回值如下。

int 类型。

0:求解过程中超过函数定义区间的右边界,程序中止运行,退出。

1:正常返回。

6. 函数说明

(1) 算法一可用来确定一元函数 $f(x)$ 的极小值所在区间,也可用来确定 $f(x)$ 的极大值所在区间。当要确定 $f(x)$ 的极大值所在区间时,只需确定 $-f(x)$ 的极小值所在区间,当确定了 $-f(x)$ 的极小值和极小值点后,其所得到的 $-f(x)$ 极小值的负值就是 $f(x)$ 的极大值,而所得到的 $-f(x)$ 的极小值点就是 $f(x)$ 的极大值点。

(2) 利用算法二与本章 14.2.7 节和 14.2.8 节求解一元函数极小值的算法相结合,就既可求解单峰目标函数 $f(x)$ 的极值,也可求解多峰目标函数的极值。对于后者,也即将构成"谷峰法",用来求解闭区间上定义的多峰一元连续函数的诸个极小值点(包括极小值)和极大值点(包括极

大值),进而可以求出该一元函数的最小值点(包括最小值)和最大值点(包括最大值)。

(3) 本函数要调用计算目标函数值的函数 FunctionValue(),由用户自己编写,其形式如下:

```
template <class _Ty>
_Ty FunctionValue(_Ty x)
{
  _Ty y;

  y=f(x)的表达式;

  return(y);
}
```

(4) 除参数 sign 外,输入、输出的数据应为浮点数类型。

7. 示例

参阅 14.2.7 和 14.2.8 节示例。

8. 示例结果

参阅 14.2.7 节和 14.2.8 节示例结果。

14.2.7 一元极小值黄金分割法

1. 功能

用黄金分割法求一元目标函数 $f(x)$的极小值和极小值点。

2. 算法

设已知初始三点 a,b,c,且 $f(b)<f(a)$,$f(b)<f(c)$,b 在 a 和 c 之间,不妨设 $a<b<c$。

① (a,c)中按黄金比率再选择一点 $d\neq b$,且若 $b-a>c-b$,则将点 d 选在(a,b)中,否则选在(b,c)中,此时为说明算法,不妨设 $b-a>c-b$,则 d 在(a,b)中,计算 $f(b)$、$f(d)$。

② 检验区间长度是否已很小,即若

$$\left|\frac{c-a}{|b|+|d|}\right|<\varepsilon$$

则转④,否则转③。

③ $f(b)<f(d)$,将 a,b,c 分别看作一组新的 a,b,c,转①,否则将 a,d,b 分别看作一组新的 a,b,c,转①。

④ 若 $f(b)<f(d)$,则将 $f(b)$看作极小值的近似值,b 为极小值点,否则将 d 作为极小值点,$f(b)$作为极小值。

3. 函数原型

```
//一元函数极小值的黄金分割法
template <class _Ty>
_Ty ExtremumGold1D(_Ty ax, _Ty bx, _Ty cx, int sign, _Ty tol, _Ty& xmin);
```

4. 参数

ax:_Ty 类型引用,开始时输入存放初始区间端点,调用后存放并输出极小值点所在区间的一个端点。

bx:_Ty 类型引用,开始时输入存放初始区间端点,调用后存放并输出极小值点的初始近似。

cx:_Ty 类型引用，存放并输出极小值点所在区间的另一个端点。

sign:int 类型引用，求一元函数 $f(x)$极小值或极大值标志，当 sign 为 1 时，求目标函数的极小值；当 sign 为－1 时，求目标函数的极大值。

tol:_Ty 类型，一元搜索精度。

xmin:_Ty 类型引用，开始时输入存放极小值点的横坐标。

5. 返回值

_Ty 类型。

返回求到函数的极小值。

6. 函数说明

(1) 本函数可求一元函数 $f(x)$的极小值，当与本章 14.2.6 节确定一元函数 $f(x)$极小值点所在区间算法的 MinimizerInterval()函数结合，既可用来求解单峰目标函数 $f(x)$的极值，也可求解多峰目标函数的极值，也即可构成“谷峰法”，用来求解闭区间上定义的多峰一元连续函数的诸个极小值点(包括极小值)和极大值点(包括极大值)，进而可以求出该一元函数的最小值点(包括最小值)和最大值点(包括最大值)。

(2) 本函数与 MinimizerInterval()函数算法一结合使用时，只能求目标函数的极小值及极小值点，这时其参数 sign 只能取值为 1；当与 MinimizerInterval()函数算法二结合使用时，可求目标函数的极大值和极小值及相应的极大值点和极小值点，故参数 sign 可取值 1 或－1。sign 的初始取定后，以后的取值由程序自动生成。

(3) 本函数要调用计算目标函数值的函数 FunctionValue()，由用户自己编写，其形式如下：

```
template <class _Ty>
_Ty FunctionValue(_Ty x)
{
  _Ty y;

  y= f(x)的表达式;

  return(y);
}
```

(4) 除参数 sign 外，输入、输出的数据应为浮点数类型。

7. 示例

示例一

用 MinimizerInterval()函数的算法一与黄金分割法计算目标函数

$$f(x) = x\sin(x)$$

在区间$[\pi, 2\pi]$内的极值点与极值点处的函数值。取初始点 $ax=3.2$，$bx=3.21$。

示例二

用 MinimizerInterval()函数算法二与黄金分割法计算目标函数

$$f(x) = x\sin(x)$$

在区间$[0, 4\pi]$内的各极值点与极值点处的函数值，此函数如图 14.1 所示。取初始点 $ax=0.1$，$bx=0.11$，右边界点 $b=14.56$。

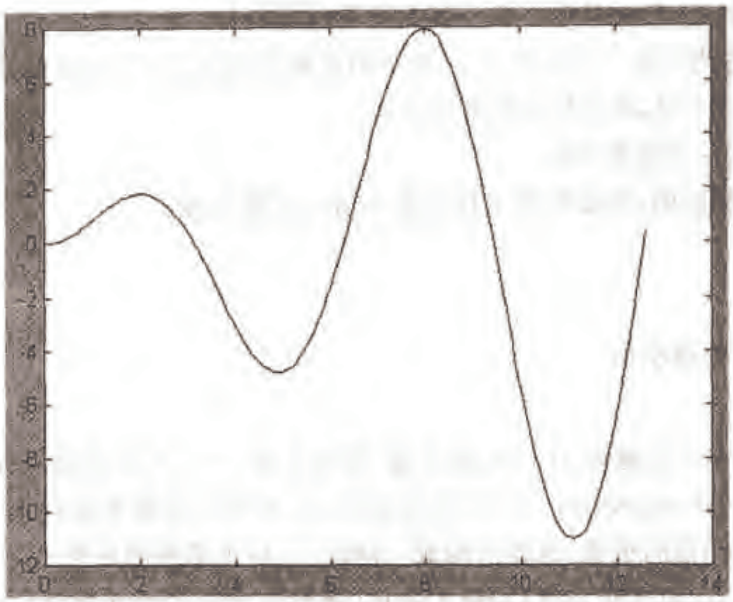


图 14.1　定义于[0,4π]上的一元函数 $f(x)=x\sin(x)$

示例一与示例二的程序合为一个程序，其主函数及计算 $f(x)$函数值的示例程序如下：

```
//ExtremumGold1D.CPP 黄金分割法求一元函数极值

#include <iostream>              //模板类 iostream 输入、输出流标准头文件
#include "Extremum.h"            //极值头文件
using namespace std;             //名字空间

void main()
{
  double fextremum,ax,bx,cx,fa,fb,fc,tol = DOUBLEERROR;
  double xmin, b = 14.56;
  int ret, sign = 1;

   cout.setf(ios::fixed|ios::left);
  cout.precision(6);

  cout<<"\t\t\t\tExample 1"<<endl;
  ax = 3.2;                  //初始点
   bx = ax + 0.01;
  cout<<" ax\t\tbx \t cx\t\tfa \t fb\t\tfc"<<endl;
  cout.width(12); cout<<ax;
  cout.width(12); cout<<bx<<endl;
   MinimizerInterval(ax, bx, cx, fa, fb, fc);
  cout.width(12); cout<<ax;
  cout.width(12); cout<<bx;
  cout.width(12); cout<<cx;
  cout.width(12); cout<<fa;
  cout.width(12); cout<<fb;
```

```
    cout.width(12); cout<<fc<<endl;

    fextremum = ExtremumGold1D(ax, bx, cx, sign, tol, xmin);
    cout<<" xmin = "<<xmin<<" \t fmin = "<<fextremum<<endl;
    cout<<endl;

    cout<<"\t\t\t\tExample 2"<<endl;
    xmin = 0.1;
    while(1)
    {
       sign = 1;
       ax = xmin + tol;
       bx = ax + 0.01;
       cout<<" ax\t\tbx \t cx\t\tfa \t fb\t\tfc"<<endl;
       cout.width(12); cout<<ax;
       cout.width(12); cout<<bx<<endl;
       if(ret = MinimizerInterval(ax, bx, cx, sign, b, fa, fb, fc) = = 0) break;
       cout.width(12); cout<<ax;
       cout.width(12); cout<<bx;
       cout.width(12); cout<<cx;
       cout.width(12); cout<<fa;
       cout.width(12); cout<<fb;
       cout.width(12); cout<<fc<<endl;

       if(bx > b) break;
       fextremum = ExtremumGold1D(ax, bx, cx, sign, tol, xmin);
       if(sign = = 1)
           cout<<" xmin = "<<xmin<<" \t fmin = "<<fextremum<<endl;
       else
           cout<<" xmax = "<<xmin<<" \t fmax = "<<fextremum<<endl;
       cout<<endl;
    }
}

double FunctionValue(double x)
{
  double y;
  y = x * sin(x);
  return y;
}
```

8. 示例结果

Example 1

ax	bx	cx	fa	fb	fc
3.150000	3.160000				

```
3.893950        4.363735      6.184545      -2.660975    -4.101185    -0.609058
    xmin = 4.913180      fmin = -4.814470

                              Example 2
    ax          bx          cx          fa          fb          fc
0.100000        0.110000
1.313735        2.073865    2.380850    -1.270568   -1.816928   -1.641500
  xmax = 2.028758 fmax = 1.819706

    ax          bx          cx          fa          fb          fc
2.028758        2.038758
4.002622        5.137457    5.232537    -3.036045   -4.680279   -4.540512
  xmin = 4.913180 fmin = -4.814470

    ax          bx          cx          fa          fb          fc
4.913180        4.923180
6.887045        8.116959    8.493019    -3.910628   -7.837900   -6.817094
  xmax = 7.978666 fmax = 7.916727

    ax          bx          cx          fa          fb          fc
7.978666        7.988666
9.952530        11.182444   11.637077   -5.012022   -10.987764  -9.323599
  xmin = 11.085538 fmin = -11.040708

    ax          bx          cx          fa          fb          fc
11.085538       11.095538
  The MinimizerInterval exceed right border. Program is over!
```

9. 示例说明

示例中使用了确定一元函数 $f(x)$极小值点所在区间算法的 MinimizerInterval()函数中的两个算法。另外可知，在算法一中，只能够对目标函数的单峰(谷)求出其极值和极值点；而算法二可以求出多峰目标函数的多个极值和极值点。

示例中输出语句提示信息中的 xmin、xmax、fmin、fmax 分别表示目标函数的极小值点、极大值点、极小值和极大值。

14.2.8 一元不用导数极小值布伦特法

1. 功能

采用不使用导数的布伦特(Brent)法求一元目标函数 $f(x)$的极小值和极小值点。

2. 算法

求解 $\min\limits_{x\in[a,b]} f(x)$，且假设$[a,b]$中有 $f(x)$的局部极小值点 $x\in(a,b)$，且 $f(x)<f(a)$，$f(x)<f(b)$。求极小值点的思想和黄金分割算法类似，即通过选取试探点，使包含极小点的区间不断缩短，当区间长度小到一定程度时，区间上各点的函数值均接近极小值，任意一点均可作为极小值点的近似。具体计算步骤如下。

(1) $k=0, v=w=x, f(v)=f(w)=f(x)$

(2) 计算当前区间的中点 $x_m=\frac{a+b}{2}$，若 $|b-a|<\varepsilon$，则 x 可作为近似极小值点，否则转(3)。

(3) 出现下面几种情况：

① 若点 x,w,v 共线，转向(4)。

② 若新找到的近似点 u 与当前函数的最小值点 x 之间满足 $|u-x|\leqslant\frac{1}{2}\max\{|a-b|,|b-x|\}$，转(4)。

③ 当 $|u-x|<\frac{1}{2}\max\{|a-b|,|b-x|\}$ 时，用点 $(x,f(x)),(w,f(w)),(v,f(v))$ 作抛物线插值，求出其极小值点 u，用它作为函数极小值点的新的近似，但要排除下面情形：若前面的极小值点 u 距当前包含极小值的区间的任意一个端点很近，则上面求出的 u 无用，将 x 作微小变化作为新的 u 而代替上面计算出的抛物线插值的极小值点 u 转向(5)。

(4) 按黄金分割选取点 u，且 u 选在两区间 $[a,x]$ 和 $[b,x]$ 长度较大的一个之中。若用黄金分割选取的 u 相对于 x 的改变量小于 $\varepsilon|x|$，则用 $\varepsilon|x|$ 代替前面的改变量。

(5) 计算 $f(u)$，由 u,x,a,b,w,v 及 $f(u),f(x),f(w),f(v)$ 将当前的 u,x,a,b,w,v 按其定义作相应改变，得到一组新的 $u,x,a,b,w,v,k\Leftarrow k+1$，进行下一次迭代，即转(2)。

点 u,x,a,b,w,v 的定义如下。

(a,b)：去掉对应的极小值点最远的点后得到的新的包含极小值点的区间。

x：其函数值 $f(x)$ 是当前求出的几个函数值中的最小值。

w：其函数值 $f(w)$ 是当前求出的几个函数值中的第二小函数值，即 $f(w)>f(x)$，但比其他算出的函数值小。

v：是前一次 w 的值。

u：是最新求函数值的点。

上面方法的结束标志有两个，一是已经可以确定的极小值点，一定在一个长度小于指定要求的区间之内，迭代停止；二是迭代次数超过指定的最大允许值时，停止迭代，表示迭代失败。

3. 函数原型

```
//一元函数极小值的不用导数布伦特(Brent)法
template <class _Ty>
_Ty ExtremumBrentNonDerivative1D(_Ty ax, _Ty bx, _Ty cx, int sign,
                                 _Ty tol, _Ty& xmin);
```

4. 参数

ax：_Ty 类型引用，为含极小值点的区间端点。

bx：_Ty 类型引用，它在 ax,cx 之间，且满足 $f(bx)<f(ax),f(bx)<f(cx)$。

cx：_Ty 类型引用，存放并输出极小值点所在区间的另一个端点。

sign：int 类型引用，求一元函数 $f(x)$ 极小值或极大值标志，当 sign 为 1 时，求目标函数的极小值；当 sign 为 −1 时，求目标函数的极大值。

tol：_Ty 类型，为变量 xmin 的相对误差精度。

xmin：_Ty 类型引用，存放极小值点的近似值最终结果。

5. 返回值

_Ty 类型。

返回求到函数的极小值。

6. 函数说明

(1) 本函数可求一元函数 $f(x)$的极小值，当与本章 14.2.6 节确定一元函数 $f(x)$极小值点所在区间算法的 MinimizerInterval()函数结合，既可用来求解单峰目标函数 $f(x)$的极值，也可求解多峰目标函数的极值，即构成"谷峰法"，用来求解闭区间上定义的多峰一元连续函数的诸个极小值点(包括极小值)和极大值点(包括极大值)，进而可以求出该一元函数的最小值点(包括最小值)和最大值点(包括最大值)。

(2) 本函数与 MinimizerInterval()函数算法结合使用时，只能求目标函数的极小值及极小值点，这时其参数 sign 只能取值为 1；当与 MinimizerInterval()函数算法二结合使用时，可求目标函数的极大值和极小值及相应的极大值点和极小值点，故参数 sign 可取值 1 或 −1。sign 的初始取定后，以后的取值由程序自动生成。

(3) 本函数要调用计算目标函数值的函数 FunctionValue()，由用户自己编写，其形式如下：

```
template <class _Ty>
_Ty FunctionValue(_Ty x)
{
  _Ty y;
  y = f(x) 的表达式;

  return(y);
}
```

(4) 除参数 sign 外，输入输出的数据应为浮点数类型。

7. 示例

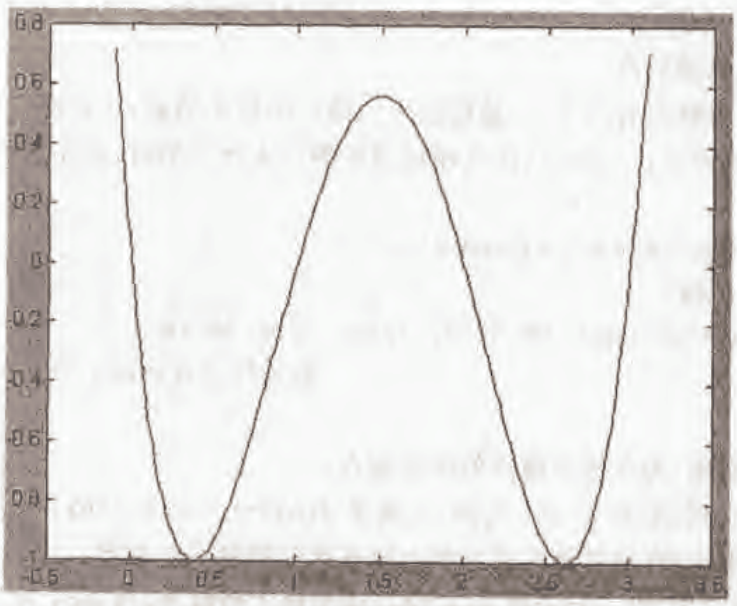


图 14.2 定义于[0,4]上的一元函数 $f(x)=(x-1)(x-2)(x-3)(x-4)$

示例一

用 MinimizerInterval()函数的算法一与不用导数的布伦特法计算目标函数

$$f(x)=x(x-1)(x-2)(x-3)(x-4)$$

在区间[1.0,2.0]内的极值点与极值点处的函数值。取初始点 $ax=1.15$，$bx=1.16$。

示例二

用 MinimizerInterval()函数算法二与不用导数的布伦特法，采用谷峰法计算目标函数

$$f(x) = x(x-1)(x-2)(x-3)(x-4)$$

在区间[0.0,4.0]内的各极值点与极值点处的函数值。取初始点 $ax=0.0$，$bx=0.01$，右边界点 $b=4.56$。

示例一与示例二的程序合为一个程序，其主函数及计算 $f(x)$函数值的示例程序如下：

```
//ExtremumBrentNonDerivative1D.CPP 不用导数的布伦特法求一元函数极小值

#include <iostream>              //模板类 iostream 输入、输出流标准头文件
#include "Extremum.h"            //极值头文件
using namespace std;             //名字空间

void main()
{
  double fextremum,ax,bx,cx,fa,fb,fc,tol = DOUBLEERROR;
  double xmin, b = 4.56;
  int ret, sign = 1;

  cout.setf(ios::fixed|ios::left);
  cout.precision(6);

  cout<<"\t\t\t\tExample 1"<<endl;
  ax = 1.15;                     //初始点
   bx = ax + 0.01;
  cout<<" ax\t\tbx \t cx\t\tfa \t fb\t\tfc"<<endl;
  cout.width(12); cout<<ax;
  cout.width(12); cout<<bx<<endl;
   MinimizerInterval(ax, bx, cx, fa, fb, fc);
  cout.width(12); cout<<ax;
  cout.width(12); cout<<bx;
  cout.width(12); cout<<cx;
  cout.width(12); cout<<fa;
  cout.width(12); cout<<fb;
  cout.width(12); cout<<fc<<endl;

  fextremum = ExtremumBrentNonDerivative1D(ax, bx, cx, sign, tol, xmin);
  cout<<" xmin = "<<xmin<<" \t fmin = "<<fextremum<<endl;
  cout<<endl;

  cout<<"\t\t\t\tExample 2"<<endl;
  xmin = 0.0;          //初始点

  while(1)
  {
```

```
        sign = 1;
        ax = xmin + tol; //ax = 0.381966;
         bx = ax + 0.01;
        cout<<" ax\t\tbx \t cx\t\tfa \t fb\t\tfc"<<endl;
        cout.width(12); cout<<ax;
        cout.width(12); cout<<bx<<endl;
         if(ret = MinimizerInterval(ax, bx, cx, sign, b, fa, fb, fc) == 0) break;
        cout.width(12); cout<<ax;
        cout.width(12); cout<<bx;
        cout.width(12); cout<<cx;
        cout.width(12); cout<<fa;
        cout.width(12); cout<<fb;
        cout.width(12); cout<<fc<<endl;

        if(bx > b) break;
         fextremum = ExtremumBrentNonDerivative1D(ax, bx, cx, sign, tol, xmin);
        if(sign == 1)
           cout<<" xmin = "<<xmin<<" \t fmin = "<<fextremum<<endl;
        else
           cout<<" xmax = "<<xmin<<" \t fmax = "<<fextremum<<endl;
        cout<<endl;
    }

double FunctionValue(double x)
{
    double y;
    y = x*(x-1)*(x-2)*(x-3)*(x-4);

    return y;
}
```

8. 示例结果

Example 1

ax	bx	cx	fa	fb	fc
1.150000	1.160000				
1.176180	1.466239	1.935564	-0.879189	-1.418024	-0.256406

xmin = 1.456088 fmin = -1.418697

Example 2

ax	bx	cx	fa	fb	fc
0.000000	0.010000				
0.026180	0.246090	0.601912	-0.594681	-3.363990	-2.729895

xmax = 0.355567 fmax = 3.631432

ax	bx
0.355567	0.365567

```
0.809173        1.099517    2.894187    1.285407    -0.543136   0.573585
  xmin = 1.456088 fmin = -1.418697

    ax          bx          cx          fa          fb          fc
1.456088        1.466088
2.200037        2.601427    2.669823    -0.760448   -1.396673   -1.311502
  xmax = 2.543912 fmax = 1.418697

    ax          bx          cx          fa          fb          fc
2.543912        2.553912
3.287862        3.757648    3.797512    -1.985917   -3.344268   -3.083752
  xmin = 3.644433 fmin = -3.631432

    ax          bx          cx          fa          fb          fc
3.644433        3.654433
  The MinimizerInterval exceed right border. Program is over!
```

9. 示例说明

示例中使用了确定一元函数 $f(x)$ 极小值点所在区间算法的 MinimizerInterval()函数中的两个算法。另外可知，在算法一中，只能够对目标函数的单峰(谷)求出其极值和极值点；而算法二可以求出多峰目标函数的多个极值和极值点。

示例中输出语句提示信息中的 xmin、xmax、fmin 和 fmax 分别表示目标函数的极小值点、极大值点、极小值和极大值。

第 15 章　数学变换与滤波

本章介绍有关的各种数学变换与滤波算法及程序，每种算法都用C++语言以函数的形式实现。15.1 节对数学变换与滤波进行简单的说明，并给出数学变换与滤波算法程序原型，15.2 节给出了数学变换与滤波的使用示例及相关说明。

15.1　数学变换与滤波的说明与头文件

本节对数学变换与滤波进行简单的说明，数学变换与滤波算法的函数程序原型包含在头文件 Transform. h 中，函数的定义内容在 Transform. inl 文件中。

本章介绍的数学变换与滤波算法及程序包括：

- 傅里叶级数逼近；
- 快速傅里叶变换；
- 快速沃什变换；
- 五点三次平滑(曲线拟合)；
- 离散随机线性系统的卡尔曼滤波；
- $\alpha-\beta-\gamma$ 滤波。

15.1.1　数学变换与滤波的说明

本章的数学变换与滤波包括快速傅里叶变换、快速沃什变换、函数的逼近、曲线拟合及滤波等内容。

傅里叶分析在研究振动和波动现象及求解数学物理方程时是个重要的工具。它在物理上还说明：任意波形总能进行谱分解，即表现为不同频率、不同振幅的简谐波的线性叠加。快速傅里叶变换的发展为傅里叶分析在实际中的广泛应用创造了条件。

15.1.2　Transform. h

```
//Transform.h        数学变换头文件
// Ver 1.0.0.0
// 版权所有(C) 何渝，2002
// 最后修改：2002.9.6.

#ifndef _TRANSFORM_H        //避免多次编译
#define _TRANSFORM_H

#include <iostream>         //模板类输入、输出流标准头文件
#include <valarray>         //模板类数组的标准头文件
#include <Matrix.h>         //模板类矩阵头文件
```

```
#include <comm.h>          //公共头文件
using namespace std;       //名字空间

//傅里叶级数逼近
template <class _Ty>
void FourierSeriesApproach(valarray<_Ty>& f,
                           valarray<_Ty>& a, valarray<_Ty>& b);

//快速傅里叶变换
template <class _Ty>
void FourierTransform(valarray<_Ty>& pr, valarray<_Ty>& pi,
        valarray<_Ty>& fr, valarray<_Ty>& fi, int l, int il);

//快速沃什变换
template <class _Ty>
void WalshTransform(valarray<_Ty>& p, valarray<_Ty>& x);

//五点三次平滑(曲线拟合)
template <class _Ty>
void Smooth5_3(valarray<_Ty>& y, valarray<_Ty>& yy);

//离散随机线性系统的卡尔曼滤波
template <class _Ty>
int SievingKalman(matrix<_Ty>& f, matrix<_Ty>& q, matrix<_Ty>& r,
                  matrix<_Ty>& h, matrix<_Ty>& y, matrix<_Ty>& x,
                                  matrix<_Ty>& p, matrix<_Ty>& g);

//a-b-r滤波
template <class _Ty>
void SievingABR(valarray<_Ty>& x,_Ty t, _Ty a, _Ty b,
                                  _Ty r, valarray<_Ty>& y);

#include "Transform.inl"   //类及相关函数的定义头文件

#endif          // _TRANSFORM_H
```

15.2 数学变换与滤波数值计算方法

15.2.1 傅里叶级数逼近

1. 功能

根据函数 $f(x)$ 在区间$[0,2\pi]$上的 $2n+1$ 个等距点

$$x_i = \frac{2\pi}{2n+1}(i+0.5), i = 0,1,\cdots,2n$$

处的函数值 $f_i = f(x_i)$，计算傅里叶(Fourier)级数

$$f(x) = \frac{1}{2}a_0 + \sum_{k=1}^{\infty}(a_k\cos kx + b_k\sin kx)$$

的前 $2n+1$ 个系数 $a_k(k = 0,1,\cdots,n)$ 和 $b_k(k = 1,2,\cdots,n)$ 的近似值。

2. 算法

设函数 $f(x)$ 在区间$[0,2\pi]$上的 $2n+1$ 个点

$$x_i = \frac{2\pi}{2n+1}(i+0.5), i = 0,1,\cdots,2n$$

处的函数值 $f_i = f(x_i)$，计算傅里叶级数的前 $2n+1$ 个系数

$$a_k(k = 0,1,\cdots,n) \text{ 和 } b_k(k = 1,2,\cdots,n)$$

的近似值的方法如下：

对于 $k = 0,1,\cdots,n$ 作如下运算：

(1) 按下列迭代公式计算 u_1 与 u_2

$$\begin{cases} u_{2n+2} = u_{2n+1} = 0 \\ u_j = f_j + 2\cos k\theta u_{j+1} - u_{j+2}, \quad j = 2n,2n-1,\cdots,2,1 \end{cases}$$

其中 $\theta = \frac{2\pi}{2n+1}$。计算 $\cos k\theta$ 与 $\sin k\theta$ 用如下递推公式

$$\cos k\theta = \cos\theta\cos(k-1)\theta - \sin\theta\sin(k-1)\theta$$
$$\sin k\theta = \sin\theta\cos(k-1)\theta + \cos\theta\sin(k-1)\theta$$

(2) 按下列公式计算 a_k 及 b_k

$$\begin{cases} a_k = \frac{2}{2n+1}(f_0 + u_1\cos k\theta - u_2) \\ b_k = \frac{2}{2n+1}u_1\sin k\theta \end{cases}$$

3. 函数原型

```
//傅里叶级数逼近
template <class _Ty>
void FourierSeriesApproach(valarray<_Ty>& f,valarray<_Ty>& a, valarray<_Ty>& b);
```

4. 参数

f:valarray<_Ty>类型引用，长度为 $2n+1$，存放区间$[0,2\pi]$上 $2n+1$ 个等距点

$$x_i = \frac{2\pi}{2n+1}(i+0.5), i = 0,1,\cdots,2n$$

处的函数值 $f(x_i)(i = 0,1,\cdots,2n)$。

a:valarray<_Ty>类型引用，长度为 $2n+1$，返回傅里叶级数中的系数 $a_k(k = 0,1,\cdots,n)$。

b:valarray<_Ty>类型引用，长度为 $2n+1$，返回傅里叶级数中的系数和 $b_k(k = 1,2,\cdots,n)$，其中 $b_0 = 0$。

5. 返回值

本函数无返回值。

6. 函数说明

本函数的参数类型应为浮点数类型。

7. 示例

根据函数 $f(x) = x^2$ 在区间 $[0,2\pi]$ 上 101 个等距点

$$x_i = \frac{2\pi}{101}(i+0.5), i = 0,1,\cdots,100$$

处的函数值 $f_i = x_i^2$, $i = 0,1,\cdots,100$,求傅里叶级数的系数

$$a_k(k = 0,1,\cdots,50) \text{ 及 } b_k(k = 1,2,\cdots,50)$$

其中 $n = 50$ 。

主函数及示例程序如下：

```
//FourierSeriesApproach.cpp        傅里叶级数逼近

#include <iostream>                 //模板类输入、输出流标准头文件
#include <Transform.h>              //数学变换头文件
using namespace std;                //名字空间

void main(void)
{
  valarray<long double> a(51);
  valarray<long double> b(51);
  valarray<long double> f(101);

  long double h = 6.283185306 / 101.0;

  for(int i = 0; i<101; i++)
    {
      long double ld = (i + 0.5) * h;
      f[i] = ld * ld;
    }

  cout.setf(ios::fixed);            //输出数据为定点法
  cout.precision(8);                //精度 8 位

  FourierSeriesApproach(f, a, b);

  cout << endl;

  for (i = 0; i<51; i++
      cout << "a(" << i << ") = " << a[i]
                        << "\t b(" << i << ") = " << b[i] << endl;
}
```

8. 示例结果

```
a(0) = 26.31830005     b(0) = 0.00000000
a(1) = 3.60654478      b(1) = -12.68669703
a(2) = 0.60654553      b(2) = -6.33720728
a(3) = 0.05099122      b(3) = -4.21797773
a(4) = -0.14345147     b(4) = -3.15630926
```

a(5) = −0.23344921 b(5) = −2.51766077
a(6) = −0.28233532 b(6) = −2.09051745
a(7) = −0.31181049 b(7) = −1.78422920
a(8) = −0.33093934 b(8) = −1.55347010
a(9) = −0.34405228 b(9) = −1.37305836
a(10) = −0.35343012 b(10) = −1.22788419
a(11) = −0.36036685 b(11) = −1.10833165
a(12) = −0.36564094 b(12) = −1.00798948
a(13) = −0.36974353 b(13) = −0.92241857
a(14) = −0.37299687 b(14) = −0.84844756
a(15) = −0.37561953 b(15) = −0.78375026
a(16) = −0.37776397 b(16) = −0.72658153
a(17) = −0.37953917 b(17) = −0.67560631
a(18) = −0.38102471 b(18) = −0.62978578
a(19) = −0.38227977 b(19) = −0.58829929
a(20) = −0.38334914 b(20) = −0.55048986
a(21) = −0.38426717 b(21) = −0.51582516
a(22) = −0.38506058 b(22) = −0.48386917
a(23) = −0.38575043 b(23) = −0.45426123
a(24) = −0.38635345 b(24) = −0.42670028
a(25) = −0.38688311 b(25) = −0.40093296
a(26) = −0.38735029 b(26) = −0.37674437
a(27) = −0.38776394 b(27) = −0.35395097
a(28) = −0.38813139 b(28) = −0.33239488
a(29) = −0.38845873 b(29) = −0.31193951
a(30) = −0.38875105 b(30) = −0.29246591
a(31) = −0.38901261 b(31) = −0.27386993
a(32) = −0.38924701 b(32) = −0.25605985
a(33) = −0.38945731 b(33) = −0.23895448
a(34) = −0.38964609 b(34) = −0.22248155
a(35) = −0.38981558 b(35) = −0.20657643
a(36) = −0.38996767 b(36) = −0.19118098
a(37) = −0.39010398 b(37) = −0.17624263
a(38) = −0.39022593 b(38) = −0.16171365
a(39) = −0.39033472 b(39) = −0.14755039
a(40) = −0.39043138 b(40) = −0.13371280
a(41) = −0.39051681 b(41) = −0.12016385
a(42) = −0.39059177 b(42) = −0.10686915
a(43) = −0.39065692 b(43) = −0.09379656
a(44) = −0.39071281 b(44) = −0.08091585
a(45) = −0.39075991 b(45) = −0.06819841
a(46) = −0.39079860 b(46) = −0.05561697
a(47) = −0.39082919 b(47) = −0.04314535
a(48) = −0.39085193 b(48) = −0.03075828

```
a(49) = -0.39086699  b(49) = -0.01843112
a(50) = -0.39087449  b(50) = -0.00613973
```

15.2.2 快速傅里叶变换

1. 功能

用 FFT 计算离散傅里叶(Fourier)变换。

2. 算法

计算 n 个采样点

$$P = \{p_0, p_1, \cdots, p_{n-1}\}$$

的离散傅里叶变换可以归结为计算多项式

$$F(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{n-1} x^{n-1}$$

在各 n 次单位根 $1, \omega, \omega^2, \cdots, \omega^{n-1}$ 上的值，即

$$\begin{aligned}
F_0 &= p_0 + p_1 + p_2 + \cdots + p_{n-1} \\
F_1 &= p_0 + p_1\omega + p_2\omega^2 + \cdots + p_{n-1}\omega^{n-1} \\
F_2 &= p_0 + p_1\omega^2 + p_2(\omega^2)^2 + \cdots + p_{n-1}(\omega^2)^{n-1} \\
&\vdots \\
F_{n-1} &= p_0 + p_1\omega^{n-1} + p_2(\omega^{n-1})^2 + \cdots + p_{n-1}(\omega^{n-1})^{n-1}
\end{aligned}$$

其中

$$\omega = e^{-j\frac{2\pi}{n}}$$

为 n 次单位元根。

若 n 是 2 的 k 次幂，即 $n = 2^k (k > 0)$。则 $F(x)$ 可以分解为关于 x 的偶次幂和奇次幂两部分，即

$$F(x) = p_0 + p_2 x^2 + \cdots + p_{n-2} x^{n-2} + x(p_1 + p_3 x^2 + \cdots + p_{n-1} x^{n-2})$$

若令

$$\begin{aligned}
P_{even}(x^2) &= p_0 + p_2 x^2 + \cdots + p_{n-2} x^{n-2} \\
P_{odd}(x^2) &= p_1 + p_3 x^2 + \cdots + p_{n-1} x^{n-2}
\end{aligned}$$

则有

$$F(x) = P_{even}(x^2) + xP_{odd}(x^2)$$

并且有

$$F(-x) = P_{even}(x^2) - xP_{odd}(x^2)$$

由此可以看出，为了求 F 在各 n 次单位根上的值，只需求 P_{even} 和 P_{odd} 在 $1, \omega^2, \cdots, (\omega^{(n/2)-1})^2$ 上的值就可以了。

而 P_{even} 和 P_{odd} 同样可以分解成关于 x^2 的偶次幂和奇次幂两部分。以此类推，一直分解下去，最后可归结为求二次单位根 1 及 −1 上的值。

在实际计算时，可以将上述过程倒过来进行，这就是 FFT 算法。

3. 函数原型

```
//快速傅里叶变换
template <class _Ty>
void FourierTransform(valarray<complex<_Ty> >& pp, valarray<complex<_Ty> >& ff,
                                                   int l, int il);
```

4. 参数

pp：valarray<complex<_Ty> >类型引用，长度为 n。当 $l = 0$ 时，存放 n 个采样输入的

(复数)数据。返回时,其实数部分存放离散傅里叶变换的模,虚数部分存放离散傅里叶变换的幅角。当 $l=1$ 时,其实数部分存放傅里叶变换的 n 个实部,返回时存放逆傅里叶变换的模;其虚数部分存放傅里叶变换的 n 个实部,返回时存放逆傅里叶变换的幅角,其中幅角的单位为度。

ff:valarray<<complex<_Ty> >类型引用,长度为 n。当 $l=0$ 时,其实数部分返回傅里叶变换的实部,其虚数部分返回逆傅里叶变换的虚部;当 $l=1$ 时,其实数部分返回傅里叶变换的实部,其虚数部分返回逆傅里叶变换的虚部。

l:整型变量。若 $l=0$,表示要求本函数计算傅里叶变换,若 $l=1$,表示要求本函数计算逆傅里叶变换。

il:整型变量。若 $il=0$,表示不要求本函数计算傅里叶变换或逆变换的模与幅角;若 $il=1$,表示要求本函数计算傅里叶变换或逆变换的模与幅角。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数中输入点数为 n,且满足 $n=2^k$。

(2) 参数中除 l 和 il 外,其余的类型都应为复数的浮点数类型。

(3) 计算结果的返回值通过参数返回,且输入的数据要使用复数类型的格式。

7. 示例

设函数 $p(t)=e^{-t},t\geqslant 0$,取 $n=64,k=6$,周期 $T=6.4$,步长为 $h=T/n=0.1$。采样序列为

$$p_i=p((i+0.5)h),i=0,1,\cdots,63$$

计算

(1) p_i 的离散傅里叶变换 f_i,且求 f_i 的模与幅角。即取 $l=0,il=1$。

(2) f_i 的逆傅里叶变换 p_i 的模,即取 $l=1,il=1$。

主函数及示例程序如下:

```
//FourierTransform.cpp  快速傅里叶变换

#include <iostream>        //模板类输入、输出流标准头文件
#include <Transform.h>     //数学变换头文件
using namespace std;       //名字空间

void main(void)
{
  valarray< complex<long double> > p(64);
  valarray< complex<long double> > f(64);

  for(int i = 0; i < 64; i++)
    {
      p[i] = complex<long double>(exp(-0.1*(i+0.5)), 0.0);

    }
```

```
cout << "序列 Pi(i = 0,1,...,63):" << endl;

for(i = 0; i< 16; i++)
  {
    for(int j = 0; j < 4; j++)
        cout << "   " << p[4 * i + j] << "   ";
    cout << endl;
}
cout << endl;

cout.setf(ios::fixed); //输出数据为定点法
cout.precision(6);     //精度 6 位

FourierTransform(p, f, 0, 1);

cout << "Pi 的傅里叶变换 Fi(i = 0,1,...,63):" << endl;
for(i = 0; i<32; i++)
  {
    for(int j = 0; j<2; j++)
        //cout<<f[4 * i + j].real()<< "\t";     //Fi 实部
        cout << "   " << f[2 * i + j]<< "\t";
    cout << endl;
}
cout << endl;

cout << "Pi 的傅里叶变换 Fi(i = 0,1,...,63)的模:" << endl;
for(i = 0; i<16; i++)
  {
    for(int j = 0; j<4; j++)
        cout << "   " << p[4 * i + j].real() << "\t";
    cout << endl;
}
cout << endl;

cout << "Pi 的傅里叶变换 Fi(i = 0,1,...,63)的幅角(单位为度):" << endl;
for(i = 0; i<16; i++)
  {
    for(int j = 0; j<4; j++)
        cout << "   " << p[4 * i + j].imag() << "\t";
    cout << endl;
}
cout << endl;

FourierTransform(f, p, 1, 1);
```

```
    cout << "Fi 的逆傅里叶变换 Pi(i = 0,1,...,63)的模:" << endl;
    for(i = 0; i<16; i ++ )
      {
          for(int j = 0; j<4; j ++ )
              cout << "   " << f[4 * i + j]. real() << "\t";
          cout << endl;
      }
    cout << endl;
}
```

8. 示例结果

序列 Pi(i = 0,1,...,63):

(0.951229,0)	(0.860708,0)	(0.778801,0)	(0.704688,0)
(0.637628,0)	(0.57695,0)	(0.522046,0)	(0.472367,0)
(0.427415,0)	(0.386741,0)	(0.349938,0)	(0.316637,0)
(0.286505,0)	(0.25924,0)	(0.23457,0)	(0.212248,0)
(0.19205,0)	(0.173774,0)	(0.157237,0)	(0.142274,0)
(0.128735,0)	(0.116484,0)	(0.105399,0)	(0.0953692,0)
(0.0862936,0)	(0.0780817,0)	(0.0706512,0)	(0.0639279,0)
(0.0578443,0)	(0.0523397,0)	(0.0473589,0)	(0.0428521,0)
(0.0387742,0)	(0.0350844,0)	(0.0317456,0)	(0.0287246,0)
(0.0259911,0)	(0.0235177,0)	(0.0212797,0)	(0.0192547,0)
(0.0174224,0)	(0.0157644,0)	(0.0142642,0)	(0.0129068,0)
(0.0116786,0)	(0.0105672,0)	(0.0095616,0)	(0.0086517,0)
(0.00782838,0)	(0.00708341,0)	(0.00640933,0)	(0.0057994,0)
(0.00524752,0)	(0.00474815,0)	(0.0042963,0)	(0.00388746,0)
(0.00351752,0)	(0.00318278,0)	(0.0028799,0)	(0.00260584,0)
(0.00235786,0)	(0.00213348,0)	(0.00193045,0)	(0.00174675,0)

Pi 的傅里叶变换 Fi(i = 0,1,...,63):

(9.979226,0.000000)	(5.318442, - 4.739672)
(2.438648, - 3.824852)	(1.464376, - 2.867734)
(1.061103, - 2.239855)	(0.861244, - 1.818536)
(0.748900, - 1.520149)	(0.679837, - 1.298421)
(0.634482, - 1.127071)	(0.603157, - 0.990376)
(0.580650, - 0.878443)	(0.563957, - 0.784768)
(0.551251, - 0.704911)	(0.541370, - 0.635744)
(0.533549, - 0.575000)	(0.527264, - 0.520998)
(0.522149, - 0.472460)	(0.517944, - 0.428403)
(0.514456, - 0.388053)	(0.511543, - 0.350793)
(0.509098, - 0.316123)	(0.507039, - 0.283634)
(0.505301, - 0.252985)	(0.503836, - 0.223890)
(0.502604, - 0.196104)	(0.501574, - 0.169417)
(0.500723, - 0.143644)	(0.500030, - 0.118622)

(0.499482, -0.094203)	(0.499067, -0.070254)
(0.498775, -0.046648)	(0.498603, -0.023268)
(0.498546,0.000000)	(0.498603,0.023268)
(0.498775,0.046648)	(0.499067,0.070254)
(0.499482,0.094203)	(0.500030,0.118622)
(0.500723,0.143644)	(0.501574,0.169417)
(0.502604,0.196104)	(0.503836,0.223890)
(0.505301,0.252985)	(0.507039,0.283634)
(0.509098,0.316123)	(0.511543,0.350793)
(0.514456,0.388053)	(0.517944,0.428403)
(0.522149,0.472460)	(0.527264,0.520998)
(0.533549,0.575000)	(0.541370,0.635744)
(0.551251,0.704911)	(0.563957,0.784768)
(0.580650,0.878443)	(0.603157,0.990376)
(0.634482,1.127071)	(0.679837,1.298421)
(0.748900,1.520149)	(0.861244,1.818536)
(1.061103,2.239855)	(1.464376,2.867734)
(2.438648,3.824852)	(5.318442,4.739672)

Pi 的傅里叶变换 Fi(i = 0,1,...,63)的模：

9.979226	7.123926	4.536132	3.219983
2.478486	2.012167	1.694610	1.465631
1.293390	1.159588	1.053004	0.966389
0.894861	0.835016	0.784410	0.741246
0.704172	0.672157	0.644399	0.620268
0.599262	0.580979	0.565094	0.551341
0.539507	0.529414	0.520919	0.513908
0.508288	0.503987	0.500952	0.499146
0.498546	0.499146	0.500952	0.503987
0.508288	0.513908	0.520919	0.529414
0.539507	0.551341	0.565094	0.580979
0.599262	0.620268	0.644399	0.672157
0.704172	0.741246	0.784410	0.835016
0.894861	0.966389	1.053004	1.159588
1.293390	1.465631	1.694610	2.012167
2.478486	3.219983	4.536132	7.123926

Pi 的傅里叶变换 Fi(i = 0,1,...,63)的幅角(单位为度)：

0.000000	-41.706684	-57.479241	-62.949386
-64.651343	-64.658146	-63.772868	-62.364003
-60.622764	-58.657814	-56.535331	-54.297881
-51.974066	-49.583817	-47.141449	-44.657517
-42.139974	-39.594939	-37.027201	-34.440567
-31.838113	-29.222354	-26.595377	-23.958936

-21.314523	-18.663425	-16.006766	-13.345543
-10.680651	-8.012910	-5.343080	-2.671880
0.000000	2.671879	5.343080	8.012910
10.680651	13.345543	16.006766	18.663425
21.314523	23.958936	26.595377	29.222354
31.838113	34.440567	37.027201	39.594939
42.139974	44.657516	47.141449	49.583817
51.974066	54.297881	56.535331	58.657814
60.622764	62.364003	63.772868	64.658146
64.651344	62.949386	57.479241	41.706684

Fi 的逆傅里叶变换 Pi(i = 0,1,...,63)的模：

0.951229	0.860708	0.778801	0.704688
0.637628	0.576950	0.522046	0.472367
0.427415	0.386741	0.349938	0.316637
0.286505	0.259240	0.234570	0.212248
0.192050	0.173774	0.157237	0.142274
0.128735	0.116484	0.105399	0.095369
0.086294	0.078082	0.070651	0.063928
0.057844	0.052340	0.047359	0.042852
0.038774	0.035084	0.031746	0.028725
0.025991	0.023518	0.021280	0.019255
0.017422	0.015764	0.014264	0.012907
0.011679	0.010567	0.009562	0.008652
0.007828	0.007083	0.006409	0.005799
0.005248	0.004748	0.004296	0.003887
0.003518	0.003183	0.002880	0.002606
0.002358	0.002133	0.001930	0.001747

15.2.3 快速沃什变换

1. 功能

计算给定序列的沃什(Walsh)变换序列。

2. 算法

设给定序列为 $p_0, p_1, \cdots, p_{n-1}$，其中 $n = 2^k, (k \geqslant 1)$，则沃什变换定义为

$$x_i = \sum_{j=0}^{n-1} W_{ij}^{(n)} p_j, i = 0,1,\cdots,n-1$$

沃什变换可以看作是矩阵与向量的乘积，即

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = W^{(n)} \cdot \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{bmatrix}$$

其中

$$W^{(n)}=\begin{bmatrix} W_{00}^{(n)} & W_{01}^{(n)} & \cdots & W_{0,n-1}^{(n)} \\ W_{10}^{(n)} & W_{11}^{(n)} & \cdots & W_{1,n-1}^{(n)} \\ \vdots & \vdots & & \vdots \\ W_{n-1,0}^{(n)} & W_{n-1,1}^{(n)} & \cdots & W_{n-1,n-1}^{(n)} \end{bmatrix}$$

是一个特殊的矩阵，其元素值只有两个：+1 与−1。$W^{(n)}$ 中的各元素有以下递推关系：

$$W^{(1)}=W_{00}^{(1)}=1$$

$$W_{2i,j}^{(2n)}=(W_{it}^{(n)},(-1)^{i}W_{it}^{(n)}),\quad t=0,1,\cdots,n-1$$

$$W_{2i+1,j}^{(2n)}=(W_{it}^{(n)},(-1)^{i+1}W_{it}^{(n)}),\quad t=0,1,\cdots,n-1$$

$$i=0,1,\cdots,n-1$$

若利用通常的矩阵相乘的方法，计算沃什变换序列需要作 $n(n-1)$ 次加减法。本函数采用与快速傅里叶变换类似的方法，加减法总次数为 $n\log_2 n$。

3. 函数原型

```
//快速沃什变换
template <class _Ty>
void WalshTransform(valarray<_Ty& p, valarray<_Ty>& x);
```

4. 参数

p：valarray<_Ty>类型引用，长度为 n，存放长度为 $n=2^k$ 的给定输入序列。

x：valarray<_Ty>类型引用，长度为 n。返回输入序列 $p_i(i=0,1,\cdots,n-1)$ 的沃什变换序列。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数中输入序列的长度为 n，且满足 $n=2^k$。

(2) 本函数中的参数类型都应为浮点数类型。

7. 示例

设输入序列为

$$p_i=i+1,i=0,1,\cdots,7$$

计算 p_i 的沃什变换序列 $x_i(i=0,1,\cdots,7)$。其中 $k=3,n=8$。

主函数及示例程序如下：

```
//WalshTransform.cpp        快速沃什变换

#include <iostream>      //模板类输入、输出流标准头文件
#include <Transform.h>   //数学变换头文件
using namespace std;     //名字空间

void main(void)
{
  valarray<long double> p(8);
  valarray<long double> x(8);

  for(int i=0; i<8; i++) p[i] = i + 1;
```

```
    WalshTransform(p, x);

    for(i = 0; i<8; i++)
        cout << "x(" << i << ") = " << x[i] << endl;
    cout << endl;
}
```

8. 示例结果

```
x(0) = 36
x(1) = -16
x(2) = 0
x(3) = -8
x(4) = 0
x(5) = 0
x(6) = 0
x(7) = -4
```

15.2.4 五点三次平滑(曲线拟合)

1. 功能

用五点三次平滑公式对等距点上的观测数据进行平滑。

2. 算法

设已知 n 个等距点 $x_0 < y_1 < \cdots < x_{n-1}$ 上的观测(或实验)为 $y_0, y_1, \cdots, y_{n-1}$,则可以在每个数据点的前后各取两个相邻的点,用三次多项式

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

进行逼近。

根据最小二乘原理确定出系数 a_0, a_1, a_2, a_3,最后可得到五点三次平滑公式如下

$$\bar{y}_{i-2} = \frac{1}{70}(69y_{i-2} + 4y_{i-1} - 6y_i + 4y_{i+1} - y_{i+2})$$

$$\bar{y}_{i-1} = \frac{1}{35}(2y_{i-2} + 27y_{i-1} + 12y_i - 8y_{i+1} + 2y_{i+2})$$

$$\bar{y}_i = \frac{1}{35}(-3y_{i-2} + 12y_{i-1} + 17y_i + 12y_{i+1} - 3y_{i+2})$$

$$\bar{y}_{i+1} = \frac{1}{35}(2y_{i-2} - 8y_{i-1} + 12y_i + 27y_{i+1} + 2y_{i+2})$$

$$\bar{y}_{i+2} = \frac{1}{70}(-y_{i-2} + 4y_{i-1} - 6y_i + 4y_{i+1} + 69y_{i+2})$$

其中 $\bar{y}_i$ 表示 y_i 的平滑值。

对于开始两点和最后两点分别由上述第 1,2 与 4,5 公式进行平滑。本方法要求数据点数 $n \geqslant 5$。

3. 函数原型

```
//五点三次平滑(曲线拟合)
template <class _Ty>
void Smooth5_3(valarray<_Ty>& y, valarray<_Ty>& yy);
```

4. 参数

y：valarray<_Ty>类型引用，长度为 n。n 个等距观测点上的观测数据。

yy：valarray<_Ty>类型引用，长度为 n。返回 n 个等距观测点上的平滑结果。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数中给定等距观测点的个数为 n，且要求 $n \geqslant 5$ 。

(2) 本函数中的参数类型都应为浮点数类型。

7. 示例

设 9 个等距观测点上的观测数据 y 为 54.0，145.0，227.0，359.0，401.0，342.0，259.0，112.0 和 65.0，用五点三次平滑公式对此 9 个观测数据进行平滑。

主函数及示例程序如下：

```
//Smooth5_3.cpp        五点三次平滑(曲线拟合)

#include <iostream>        //模板类输入、输出流标准头文件
#include <Transform.h>     //数学变换头文件
using namespace std;       //名字空间

void main(void)
{
  long double y[9] =
  {
      54.0,145.0,227.0,359.0,401.0,342.0,259.0,112.0,65.0
  };

  valarray<long double> yd(y,9);
  valarray<long double> yy(9);

  Smooth5_3(yd, yy);

  for(int i=0; i<9; i++)
      cout << "y(" << i << ") = " << y[i]
           << "\t yy(" << i << ") = " << yy[i] << endl;

  cout << endl;
}
```

8. 示例结果

```
y(0) = 54       yy(0) = 56.8429
y(1) = 145      yy(1) = 133.629
y(2) = 227      yy(2) = 244.057
y(3) = 359      yy(3) = 347.943
y(4) = 401      yy(4) = 393.457
y(5) = 342      yy(5) = 352.029
```

```
y(6) = 259      yy(6) = 241.514
y(7) = 112      yy(7) = 123.657
y(8) = 65       yy(8) = 62.0857
```

15.2.5 离散随机线性系统的卡尔曼滤波

1. 功能

对时间离散点上的采样数据进行卡尔曼(Kalman)滤波。

2. 算法

设 n 维线性动态系统与 m 维线性观测系统由下列差分方程组描述

$$\begin{cases} \boldsymbol{X}_k = \boldsymbol{\Phi}_{k,k-1}\boldsymbol{X}_{k-1} + \boldsymbol{W}_{k-1} \\ \boldsymbol{Y}_k = \boldsymbol{H}_k\boldsymbol{X}_k + \boldsymbol{V}_k \end{cases}, k = 1,2,\cdots$$

其中，$\boldsymbol{X}_k$ 为 n 维向量，表示系统在第 k 时刻的状态。

$\boldsymbol{\Phi}_{k,k-1}$ 是一个 $n\times n$ 阶矩阵，称为系统的状态转移矩阵，它反映了系统从第 $k-1$ 个采样时刻的状态到第 k 个采样时刻的状态的变换。

$\boldsymbol{W}_k$ 是一个 n 维向量，表示在第 k 时刻作用于系统的随机干扰，称为模型噪声。为简单起见一般假设$\{\boldsymbol{W}_k\}$，$(k=1,2,\cdots)$为高斯白噪声序列，具有已知的零均值和协方差阵 $\boldsymbol{Q}_k$。

$\boldsymbol{Y}_k$ 为 m 维的观测向量。

$\boldsymbol{H}_k$ 为 $m\times n$ 阶的观测矩阵，表示了从状态量 $\boldsymbol{X}_k$ 到观测量 $\boldsymbol{Y}_k$ 的转换。

$\boldsymbol{V}_k$ 为 m 维的观测噪声，同样假设$\{\boldsymbol{V}_k\}$，$(k=1,2,\cdots)$为高斯白噪声序列，具有已知的零均值和协方差阵 $\boldsymbol{R}_k$。

经推导(推导过程略)可得到如下滤波的递推公式：

$$\boldsymbol{G}_k = \boldsymbol{P}_k\boldsymbol{H}_k^{\mathrm{T}}[\boldsymbol{H}_k\boldsymbol{P}_k\boldsymbol{H}_k^{\mathrm{T}} + \boldsymbol{R}_k]^{-1}$$

$$\tilde{\boldsymbol{X}}_k = \boldsymbol{\Phi}_{k,k-1}\tilde{\boldsymbol{X}}_{k-1} + \boldsymbol{G}_k[\boldsymbol{Y}_k - \boldsymbol{H}_k\boldsymbol{\Phi}_{k,k-1}\tilde{\boldsymbol{X}}_{k-1}]$$

$$\boldsymbol{C}_k = (1 - \boldsymbol{G}_k\boldsymbol{H}_k)\boldsymbol{P}_k$$

$$\boldsymbol{P}_{k+1} = \boldsymbol{\Phi}_{k+1,k}\boldsymbol{C}_k\boldsymbol{P}_k\boldsymbol{\Phi}_{k+1,k}^{\mathrm{T}} + \boldsymbol{Q}_k$$

式中 $\boldsymbol{Q}_k$ 为 $n\times n$ 阶的模型噪声 $\boldsymbol{W}_k$ 的协方差阵；

$\boldsymbol{R}_k$ 为 $m\times m$ 阶的观测噪声 $\boldsymbol{V}_k$ 的协方差阵；

$\boldsymbol{G}_k$ 为 $n\times m$ 阶的增益矩阵；

$\tilde{\boldsymbol{X}}_k$ 为 n 维向量，第 k 时刻经滤波后的估值；

$\boldsymbol{C}_k$ 为 $n\times n$ 阶的估计误差协方差阵。

根据上述公式，可以从 $\tilde{\boldsymbol{X}}_0 = \boldsymbol{E}\{\boldsymbol{X}_0\}$，$\boldsymbol{P}_0$(给定)出发，利用已知矩阵 $\boldsymbol{Q}_k$，$\boldsymbol{R}_K$，$\boldsymbol{H}_k$，$\boldsymbol{\Phi}_{k,k-1}$ 及 k 时刻的观测值 $\boldsymbol{Y}_k$，递推地算出每个时刻的状态估计 $\tilde{\boldsymbol{X}}_k(k = 1,2,\cdots)$。

如果线性系统是定常的，则有 $\boldsymbol{\Phi}_{k,k-1} = \boldsymbol{\Phi}$，$\boldsymbol{H}_k = \boldsymbol{H}$，即它们都是常阵；如果模型噪声 $\boldsymbol{W}_k$ 和观测噪声 $\boldsymbol{V}_k$ 都是平稳随机序列，则 $\boldsymbol{Q}_k$ 和 $\boldsymbol{R}_k$ 都是常阵。在这种情况下，常增益的离散卡尔曼滤波是渐近稳定的。

3. 函数原型

```
//离散随机线性系统的卡尔曼滤波
template <class _Ty>
int SievingKalman(matrix<_Ty>& f, matrix<_Ty>& q, matrix<_Ty>& r,
                  matrix<_Ty>& h, matrix<_Ty>& y, matrix<_Ty>& x,
```

```
matrix<_Ty>& p, matrix<_Ty>& g);
```

4. 参数

f:matrix<_Ty>类型引用,体积为 $n\times n$。系统状态转移矩阵。

q:matrix<_Ty>类型引用,体积为 $n\times n$。模型噪声 W_k 的协方差阵。

r:matrix<_Ty>类型引用,体积为 $m\times m$。观测噪声 V_k 的协方差阵。

h:matrix<_Ty>类型引用,体积为 $m\times n$。观测矩阵。

y:matrix<_Ty>类型引用,体积为 $k\times m$。观测向量序列。其中 $y(i,j)(i=0,1,\cdots,k-1;j=0,1,\cdots,m-1)$ 表示第 i 时刻的观测向量的第 j 个分量。

x: matrix<_Ty>类型引用,体积为 $k\times n$。其中 $x(0,j)(j=0,1,\cdots,n-1)$ 存放给定的初值,其余各行返回状态向量估值序列。$x(i,j)(i=0,1,\cdots,k-1;j=0,1,\cdots,n-1)$ 表示第 i 时刻的状态向量估值的第 j 个分量。

p:matrix<_Ty>类型引用,体积为 $n\times n$。存放初值 P_0,返回时存放最后时刻的估计误差协方差阵。

g:matrix<_Ty>类型引用,体积为 $n\times m$。返回最后时刻的稳定增益矩阵。

5. 返回值

int 类型。

0:求增益矩阵过程中求逆失败,输出提示信息"Error! No inversion!"。

非 0:正常返回。

6. 函数说明

(1) 本函数中动态系统的维数为 n。

(2) 本函数中观测系统的维数为 m。

(3) 本函数中观测序列的长度为 k。

(4) 本函数中的参数类型都应为浮点数类型。

(5) 本函数要调用全选主元高斯－约当(Gauss－Jordan)消去法对实矩阵求逆的函数 MatrixInversionGS(),请参看第 5 章 5.5.1 节。

7. 示例

设信号源运动方程为 $s(t)=5-2t+3t^2+v(t)$,其中 $v(t)$ 是一个均值为零、方差为 0.25 的高斯白噪声。

状态向量为 $\boldsymbol{X}_k=(s,s',s'')^{\mathrm{T}}$,并取初值

$$\boldsymbol{X}_0=(0.0,0.0,0.0)^{\mathrm{T}}$$

状态转移矩阵为

$$\boldsymbol{F}=\boldsymbol{\Phi}_{k,k-1}=\begin{bmatrix}1 & T & \frac{T^2}{2}\\ 0 & 1 & T\\ 0 & 0 & 1\end{bmatrix}$$

其中 T 为采样间隔,在本例中取 $T=0.05$,即

$$\boldsymbol{F}=\begin{bmatrix}1 & 0.05 & 0.00125\\ 0 & 1 & 0.05\\ 0 & 0 & 1\end{bmatrix}$$

动态系统维数为 $n=3$,观测系统维数为 $m=1$。

模型噪声协方差阵为

$$\boldsymbol{Q}=\begin{bmatrix}0.25 & 0 & 0\\ 0 & 0.25 & 0\\ 0 & 0 & 0.25\end{bmatrix}$$

观测矩阵为 $H=(1,0,0)$。

观测噪声协方差阵为 $R=0.25$。

初始估计误差协方差阵取为

$$\boldsymbol{P}_0=\begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}$$

取观测向量序列长度为 $k=200$。

对输出的状态向量序列每隔 5 个时刻打印一次。

主函数及示例程序如下：

```
//SievingKalman.cpp            离散随机线性系统的卡尔曼滤波

#include <iostream>            //模板类输入、输出流标准头文件
#include <Transform.h>         //数学变换头文件
#include <Matrix.h>            //模板类矩阵头文件
using namespace std;           //名字空间

template <class _Ty>
void SievingKalman5(matrix<_Ty>& y)
{
  _Ty t, s=65536, w=2053, v=13849, r=0;

int k = y.GetRowNum(); //观测序列的长度

  for(int i=0; i<k; i++)
  {
    t=0.0;
    for(int j=0; j<12; j++)
    {
      r=w*r+v;
      int m=r/s;
      r=r-m*s;
      t=t+r/s;
    }
    y(i,0)=0.5*(t-6.0);
  }
}

void main(void)
{
```

```
long double f[3][3] =
{
    {1.0,0.05,0.00125}, {0.0,1.0,0.05}, {0.0,0.0,1.0}
};

long double q[3][3] =
{
    {0.25,0.0,0.0}, {0.0,0.25,0.0}, {0.0,0.0,0.25}
};

long double r[1][1] = {0.25};
long double h[1][3] = {1.0,0.0,0.0};

matrix<long double> pp(3,3), xx(200,3), yy(200,1), gg(3,1);

for (int i=0; i<3; i++)
    for (int j=0; j<3; j++) pp(i,j) = 0;

for (i=0; i<200; i++)
    for (int j=0; j<3; j++) xx(i,j) = 0;

matrix<long double> ff(&f[0][0],3,3), qq(&q[0][0],3,3);
matrix<long double> rr(&r[0][0],1,1), hh(&h[0][0],1,3);

SievingKalman5(yy);

for(i=0; i<200; i++)
{
    long double t=0.05*i;
      yy(i,0)=5.0-2.0*t+3.0*t*t+yy(i,0);
}

int js = SievingKalman(ff,qq,rr,hh,yy,xx,pp,gg);

cout.setf(ios::fixed); //输出数据为定点法

if(js!=0)
{
    cout << endl;

      cout << " t          s          y          ";
      cout << "x[0]          x[1]          x[2]" << endl;

      for(int i=0; i<200; i=i+5)
```

```
        {
            long double t = 0.05 * i;
            long double s = 5.0 - 2.0 * t + 3.0 * t * t;
            cout.precision(2);  //精度 2 位
            cout << t;
            cout.precision(6);  //精度 6 位
            cout.width(12);       //宽度 12 列
            cout << s;
            cout.width(12);
            cout << yy(i,0);
            cout.width(15);
            cout << xx(i,0);
            cout.width(15);
            cout << xx(i,1);
            cout.width(15);
            cout << xx(i,2);
            cout <<endl;
        }
        cout << endl;
    }
}
```

其中函数 Sieving Kalman5()为产生均值为零、方差为 0.5^2 的高斯白噪声序列。

运行结果见 8.示例结果(其中:t 为采样时刻值,s 为真值,y 为迭加有高斯白噪声的采样值,$x[0]$,$x[1]$和 $x[2]$为状态向量各分量的估值)。

8.示例结果

t	s	y	x[0]	x[1]	x[2]
0.00	5.000000	4.453659	0.000000	0.000000	0.000000
0.25	4.687500	5.794601	5.508562	0.076068	0.004848
0.50	4.750000	4.762497	4.728298	0.021726	0.005155
0.75	5.187500	5.138596	5.104189	0.130927	0.030762
1.00	6.000000	6.204147	5.942524	0.510189	0.155950
1.25	7.187500	5.740402	6.217366	0.581884	0.151985
1.50	8.750000	9.028610	8.742496	2.162990	0.778554
1.75	10.687500	10.350021	10.094162	2.976066	1.061182
2.00	13.000000	13.485886	13.031065	5.002371	1.862898
2.25	15.687500	16.217453	16.183207	7.107064	2.645101
2.50	18.750000	19.325974	19.070838	8.714901	3.103052
2.75	22.187500	22.092697	21.814567	9.951667	3.337508
3.00	26.000000	25.798874	25.768091	12.121047	4.005074
3.25	30.187500	30.225754	30.215352	14.370641	4.616506
3.50	34.750000	34.654587	34.696894	16.270597	4.993038
3.75	39.687500	40.366623	40.188105	18.794563	5.638742
4.00	45.000000	44.143112	44.485784	19.629474	5.350010

4.25	50.687500	50.765305	50.750243	22.120253	5.910251
4.50	56.750000	57.514450	57.186680	24.303404	6.256663
4.75	63.187500	63.171799	63.148022	25.586973	6.129899
5.00	70.000000	69.518600	69.681649	27.022290	6.060447
5.25	77.187500	77.336105	77.305020	29.220523	6.404718
5.50	84.750000	84.905563	85.119379	31.139135	6.566711
5.75	92.687500	92.008224	92.315766	31.966971	6.148932
6.00	101.000000	101.925339	101.512776	34.554212	6.692876
6.25	109.687500	109.438156	109.422504	35.323155	6.256643
6.50	118.750000	118.327927	118.296357	36.758090	6.203206
6.75	128.187500	127.375900	127.811735	38.369788	6.198350
7.00	138.000000	137.863327	137.812528	40.128671	6.300167
7.25	148.187500	148.071457	148.002102	41.666275	6.282102
7.50	158.750000	158.281540	158.346306	42.946108	6.125851
7.75	169.687500	169.274826	169.188810	44.360284	6.052940
8.00	181.000000	180.832565	180.913766	46.245965	6.203452
8.25	192.687500	192.736008	192.655090	47.798804	6.212418
8.50	204.750000	204.766403	204.749505	49.323563	6.208678
8.75	217.187500	218.205002	217.801243	51.433197	6.503293
9.00	230.000000	229.333054	229.440738	51.664383	5.818590
9.25	243.187500	242.431808	242.840205	53.359474	5.908909
9.50	256.750000	255.782516	255.952688	54.381619	5.659765
9.75	270.687500	270.666428	270.645275	56.656424	6.060824

15.2.6 $\alpha-\beta-\gamma$ 滤波

1. *功能*

对等间隔的量测数据进行滤波估值。

2. *算法*

设一个过程的量测数据为 $X^*(t)$ 且

$$X^*(t) = X(t) + \eta(t)$$

其中：$X(t)$ 为有用信号的准确值，$\eta(t)$ 是均值为零的白噪声过程，即有

$$E\{\eta(t)\} = 0$$

$$E\{\eta(t)\eta(z)\} = \gamma\delta(t-z)$$

采用 $\alpha-\beta-\gamma$ 滤波方法对量测数据序列 X^* 进行估值的计算公式如下。

由上时刻对本时刻的一步预测估值公式为

$$\tilde{X}_{n+1/n} = \tilde{X}_n + \tilde{X}'_n T + \tilde{X}''_n (T^2/2)$$

$$\tilde{X}'_{n+1/n} = \tilde{X}'_n + \tilde{X}''_n T$$

$$\tilde{X}''_{n+1/n} = \tilde{X}''_n$$

本时刻的滤波估值公式为

$$\tilde{X}''_{n+1} = \tilde{X}_{n+1/n} + \alpha(\tilde{X}^*_{n+1} + \tilde{X}_{n+1/n})$$

$$\tilde{X}'_{n+1} = \tilde{X}'_{n+1/n} + \frac{\beta}{T}(\tilde{X}^*_{n+1} + \tilde{X}_{n+1/n})$$

$$\tilde{X}''_{n+1} = \tilde{X}''_{n+1/n} + \frac{2\gamma}{T^2}(\tilde{X}^*_{n+1} + \tilde{X}_{n+1/n})$$

其中：$\tilde{X}^{*}_{n+1}$为本时刻的量测值；

$\tilde{X}_{n+1/n}$为上时刻对本时刻的位置一步预测估值；

$\tilde{X}'_{n+1/n}$为上时刻对本时刻的速度一步预测估值；

$\tilde{X}''_{n+1/n}$为上时刻对本时刻的加速度一步预测估值；

$\tilde{X}_{n+1}$为本时刻的位置滤波估值；

$\tilde{X}'_{n+1}$为本时刻的速度滤波估值；

$\tilde{X}''_{n+1}$为本时刻的加速度滤波估值；

T为采样间隔；

α、β、γ为滤波器结构的参数。

3. 函数原型

```
//a-b-r滤波
template <class _Ty>
void SievingABR(valarray<_Ty>& x, _Ty t, _Ty a, _Ty b,_Ty r, valarray<_Ty>& y);
```

4. 参数

x：valarray<_Ty>类型引用，长度为n。n个等间隔点上的量测值。

t：_Ty类型。采样周期。

a：_Ty类型。滤波器结构参数α。

b：_Ty类型。滤波器结构参数β。

r：_Ty类型。滤波器结构参数γ。

y：valarray<_Ty>类型引用，长度为n。返回n个等间隔点上的滤波估值。

5. 返回值

本函数无返回值。

6. 函数说明

(1) 本函数中量测数据的点数为n。

(2) 本函数中的参数类型都应为浮点数类型。

7. 示例

设信号为$z(t)=3t^2-2t+5$，迭加上一个均值为零、方差为0.5^2的正态分布白噪声后，以周期$T=0.04$采样250个点。

取$\alpha=0.271$，$\beta=0.0285$，$\gamma=0.0005$对此250个采样点进行滤波估值。

主函数及示例程序如下：

```
//SievingABR.cpp            a-b-r滤波

#include <iostream>       //模板类输入、输出流标准头文件
#include <Transform.h>    //数学变换头文件
using namespace std;      //名字空间

template <class _Ty>
void SievingABR5(valarray<_Ty>& y)
{
  _Ty s=65536.0, w=2053.0, v=13849.0, r=0.0;
```

```
    int n = y.size();

      for(int i = 0; i<n; i ++ )
      {
        _Ty t = 0.0;
          for(int j = 0; j<12; j ++ )
          {
            r = w * r + v;
            int m = r/s;
            r = r - m * s;
            t = t + r/s;
       }
        y[i] = 0.5 * (t - 6.0);
    }
}

void main(void)
{
    valarray<long double> x(250), y(250), z(250);
    long double a = 0.271, b = 0.0285, r = 0.0005, dt = 0.04;

    SievingABR5(y);
    for(int i = 0; i<250; i ++ )
    {
        long double t = (i + 1) * dt;
          z[i] = 3.0 * t * t - 2.0 * t + 5.0;
          x[i] = z[i] + y[i];
    }

    SievingABR(x,dt,a,b,r,y);

    for(i = 0; i<250; i = i + 5)
    {
        long double t = (i + 1) * dt;

        cout.setf(ios::fixed);      //输出数据为定点法

        cout.precision(2);          //精度 2 位
        cout << "t = " << t;
        cout.precision(6);          //精度 6 位
        cout.width(12);             //宽度 12 列
        cout << "x(t) = " << x[i];
        cout.width(12);
        cout << "y(t) = " << y[i];
```

```
        cout.width(12);
        cout << "z(t) = " << z[i];
        cout << endl;
    }
    cout << endl;
}
```

8. 示例结果

其中函数 SievingABR5()为产生均值为零、方差为 0.5^2 的正态分布白噪声序列。

运行结果如下。其中 t 为采样时刻值，$x(t)$为迭加上白噪声后的信号值，$y(t)$为滤波估值，$z(t)$为准信号值。

```
t = 0.04      x(t) = 4.378459      y(t) = 1.186562      z(t) = 4.924800
t = 0.24      x(t) = 5.799901      y(t) = 4.516264      z(t) = 4.692800
t = 0.44      x(t) = 4.713297      y(t) = 4.770187      z(t) = 4.700800
t = 0.64      x(t) = 4.899896      y(t) = 5.255358      z(t) = 4.948800
t = 0.84      x(t) = 5.640947      y(t) = 5.765773      z(t) = 5.436800
t = 1.04      x(t) = 4.717702      y(t) = 6.204424      z(t) = 6.164800
t = 1.24      x(t) = 7.411410      y(t) = 7.534516      z(t) = 7.132800
t = 1.44      x(t) = 8.003321      y(t) = 8.329771      z(t) = 8.340800
t = 1.64      x(t) = 10.274686     y(t) = 10.064617     z(t) = 9.788800
t = 1.84      x(t) = 12.006753     y(t) = 12.024317     z(t) = 11.476800
t = 2.04      x(t) = 13.980774     y(t) = 13.850805     z(t) = 13.404800
t = 2.24      x(t) = 15.477997     y(t) = 15.364614     z(t) = 15.572800
t = 2.44      x(t) = 17.779674     y(t) = 17.531491     z(t) = 17.980800
t = 2.64      x(t) = 20.667054     y(t) = 20.325859     z(t) = 20.628800
t = 2.84      x(t) = 23.421387     y(t) = 23.072984     z(t) = 23.516800
t = 3.04      x(t) = 27.323923     y(t) = 26.415726     z(t) = 26.644800
t = 3.24      x(t) = 29.155912     y(t) = 29.438385     z(t) = 30.012800
t = 3.44      x(t) = 33.698605     y(t) = 33.251906     z(t) = 33.620800
t = 3.64      x(t) = 38.233250     y(t) = 37.439696     z(t) = 37.468800
t = 3.84      x(t) = 41.541099     y(t) = 41.379172     z(t) = 41.556800
t = 4.04      x(t) = 45.403400     y(t) = 45.768815     z(t) = 45.884800
t = 4.24      x(t) = 50.601405     y(t) = 50.527235     z(t) = 50.452800
t = 4.44      x(t) = 55.416363     y(t) = 55.750511     z(t) = 55.260800
t = 4.64      x(t) = 59.629524     y(t) = 60.664711     z(t) = 60.308800
t = 4.84      x(t) = 66.522139     y(t) = 66.184066     z(t) = 65.596800
t = 5.04      x(t) = 70.875456     y(t) = 71.509083     z(t) = 71.124800
t = 5.24      x(t) = 76.470727     y(t) = 77.020843     z(t) = 76.892800
t = 5.44      x(t) = 82.089200     y(t) = 83.287489     z(t) = 82.900800
t = 5.64      x(t) = 89.012127     y(t) = 89.403681     z(t) = 89.148800
t = 5.84      x(t) = 95.520757     y(t) = 95.757584     z(t) = 95.636800
t = 6.04      x(t) = 101.896340    y(t) = 102.262083    z(t) = 102.364800
t = 6.24      x(t) = 108.920126    y(t) = 108.922862    z(t) = 109.332800
t = 6.44      x(t) = 116.373365    y(t) = 116.353303    z(t) = 116.540800
```

t = 6.64	x(t) = 124.037308	y(t) = 123.517264	z(t) = 123.988800
t = 6.84	x(t) = 131.693203	y(t) = 131.055153	z(t) = 131.676800
t = 7.04	x(t) = 140.622302	y(t) = 139.206584	z(t) = 139.604800
t = 7.24	x(t) = 147.105854	y(t) = 146.835943	z(t) = 147.772800
t = 7.44	x(t) = 155.425108	y(t) = 155.390342	z(t) = 156.180800
t = 7.64	x(t) = 163.861316	y(t) = 163.838607	z(t) = 164.828800
t = 7.84	x(t) = 173.695728	y(t) = 173.213933	z(t) = 173.716800
t = 8.04	x(t) = 182.709592	y(t) = 182.652434	z(t) = 182.844800
t = 8.24	x(t) = 192.684159	y(t) = 192.305708	z(t) = 192.212800
t = 8.44	x(t) = 201.400680	y(t) = 202.024078	z(t) = 201.820800
t = 8.64	x(t) = 211.640403	y(t) = 212.176612	z(t) = 211.668800
t = 8.84	x(t) = 221.684580	y(t) = 222.390628	z(t) = 221.756800
t = 9.04	x(t) = 231.814460	y(t) = 232.802927	z(t) = 232.084800
t = 9.24	x(t) = 241.811293	y(t) = 243.234468	z(t) = 242.652800
t = 9.44	x(t) = 253.456329	y(t) = 254.299630	z(t) = 253.460800
t = 9.64	x(t) = 265.030818	y(t) = 265.446445	z(t) = 264.508800
t = 9.84	x(t) = 275.816011	y(t) = 276.466769	z(t) = 275.796800

第16章 特殊函数

本章介绍常用特殊函数的数值算法及程序，每种算法都用C++语言以函数的形式实现。16.1节介绍特殊函数的基本概念及特殊函数算法程序原型，16.2节给出了计算各特殊函数的使用示例及相关说明。

16.1 特殊函数与头文件

本节介绍特殊函数基本概念及特殊函数算法的函数程序原型，各程序原型包含在头文件SpecialFunction.h中，函数的定义内容在SpecialFunction.inl文件中。

本章介绍的特殊函数包括：

- 伽马(Camma)函数；
- 不完全伽马(Incomplete Camma)函数；
- 误差(Error)函数；
- 第一类整数阶贝塞耳(Bessel)函数；
- 第二类整数阶贝塞耳(Bessel)函数；
- 变型第一类整数阶贝塞耳(Bessel)函数；
- 变型第二类整数阶贝塞耳(Bessel)函数；
- 不完全贝塔(beta)函数；
- 正态分布函数；
- t-分布函数；
- χ^2-分布函数；
- F-分布函数；
- 正弦积分函数；
- 余弦积分函数；
- 指数积分函数；
- 第一类椭圆积分；
- 第二类椭圆积分。

16.1.1 特殊函数

特殊函数一般是指某类微分方程的解，但又不能用初等函数的有限形式表示的函数。这类函数在应用中是常见的，如贝塞耳(Bessel)函数等；另外一些是由特定形式的积分所定义的函数，如伽马(Camma)函数等。还有从函数的周期性的角度来考虑的所谓椭圆函数，这类函数与微分方程无关。本章介绍了若干种这些特殊的函数。

16.1.2 SpecialFunction.h

```
//SpecialFunction.h                特殊函数头文件
//Ver1.0.0.0
//版权所有(C)何渝,2002
//最后修改:2002.9.6

#ifndef_SPECIALFUNCTION_H     //避免多次编译
#define_SPECIALFUNCTION_H

#include <iostream>           //模板类输入、输出流标准头文件
#include <comm.h>             //公共头文件
using namespace std;          //名字空间

//伽马函数
template <class_Ty>
_Ty GammaFunction(_Ty x);

//不完全伽马函数
template <class_Ty>
_Ty IncompleteGammaFunction(_Ty a, _Ty x);

//误差函数
template <class _Ty>
_Ty ErrorFunction(_Ty x);

//第一类整数阶贝塞尔函数
template <class_Ty>
_Ty IntegerBessel1stFunction(int n,_Ty x);

//第二类整数阶贝塞尔函数
template <class_Ty>
_Ty IntegerBessel2ndFunction(int n,_Ty x);

//变型第一类整数阶贝塞尔函数
template <class_Ty>
_Ty TransformativeIntegerBessel1stFunction(int n,_Ty x);

//变型第二类整数阶贝塞尔函数
template <class_Ty>
_Ty TransformativeIntegerBessel2ndFunction(int n,_Ty x);

//不完全贝塔函数
template <class_Ty>
```

```
_Ty IncompleteBetaFunction(_Ty a, _Ty b,_Ty x);

//正态分布函数
template <class _Ty>
_Ty NormalDistributionFunction(_Ty a, _Ty d, _Ty x);

//t-分布函数
template <class _Ty>
_Ty tDistributionFunction(_Ty t, int n);

//X^2-分布函数
template <class _Ty>
_Ty X2DistributionFunction(_Ty x, int n);

//F-分布函数
template <class _Ty>
_Ty FDistributionFunction(_Ty f, int n1, int n2);

//正弦积分
template <class _Ty>
_Ty SineIntegralFunction(_Ty x);

//余弦积分
template <class _Ty>
_Ty CosineIntegralFunction(_Ty x);

//指数积分
template <class _Ty>
_Ty ExponentIntegralFunction(_Ty x);

//第一类椭圆积分
template <class _Ty>
_Ty Ellipse1stIntegral(_Ty k, _Ty f);

//第二类椭圆积分
template <class _Ty>
_Ty Ellipse2ndIntegral(_Ty k, _Ty f);

#include "SpecialFunction.inl"              //类及相关函数的定义头文件

#endif              // _SPECIALFUNCTION_H
```

16.2 特殊函数计算方法

16.2.1 伽马函数

1. 功能

计算实变量 x 的伽马(Camma)函数值

$$\Gamma(x)=\int_0^{\infty}e^{-t}t^{x-1}dt(x>0)$$

2. 算法

当 $2<x\leqslant 3$ 时，$\Gamma(x)$用下列切比雪夫(Chebyshev)多项式逼近

$$\Gamma(x)=\sum_{i=0}^{10}a_i(x-2)^{10-i}$$

其中

$$\begin{aligned}&a_0=0.0000677106, \quad a_1=-0.0003442342\\&a_2=0.0015397681, \quad a_3=-0.0024467480\\&a_4=0.0109736958, \quad a_5=-0.0002109075\\&a_6=0.0742379071, \quad a_7=0.0815782188\\&a_8=0.4118402518, \quad a_9=0.4227843370\\&a_{10}=1.0\end{aligned}$$

当 $0<x\leqslant 2$ 时，利用公式

$$\Gamma(x)=\frac{1}{x}\Gamma(x+1),1<x\leqslant 2$$

或

$$\Gamma(x)=\frac{1}{x(x+1)}\Gamma(x+2),0<x\leqslant 2$$

当 $x>3$ 时，利用公式

$$\Gamma(x)=(x-1)(x-2)\cdots(x-i)\Gamma(x-i)$$

直到满足 $2<(x-i)\leqslant 3$ 为止。

利用伽马函数可以计算贝塔(Beta)函数

$$B(x,y)=B(y,x)=\int_0^1 t^{x-1}(1-t)^{y-1}dt,x,y>0$$

其计算公式为

$$B(x,y)=\frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$$

3. 函数原型

```
//伽马函数
template <class _Ty>
_Ty GammaFunction(_Ty x);
```

4. 参数

x：类型_Ty，自变量值。要求 $x>0$。若 $x\leqslant 0$，本函数返回函数值−1，且输出信息“Error * * $x<=0$!”。

5. 返回值

_Ty类型。

当输入的参数值 $x \leqslant 0$ 时，本函数返回函数值 -1，且输出信息“Error * * $x<0$!”；否则返回伽马函数值。

6. 函数说明

本函数的输入、输出数据类型应为浮点数类型。

7. 示例

计算 0.5～5.0 之间每隔 0.5 的伽马函数值以及贝塔函数值 $B(1.5,2.5)$。

主函数及示例程序如下：

```
//GammaFunction.cpp                    伽马函数

#include <iostream>                    //模板类输入、输出流标准头文件
#include <SpecialFunction.h>           //数学变换头文件
using namespace std;                   //名字空间

void main(void)
{
  double x, y;

  for(int i=1; i<11; i++)
    {
      x = 0.5 * i;
      y = GammaFunction(x);
      cout << "x = " << x << "  t GammaFunction(x) = " << y << endl;
  }

  cout << endl;

  y = GammaFunction(1.5) * GammaFunction(2.5) / GammaFunction(4.0);

  cout << "B(1.5,2.5) = " << y << endl;
  }
```

8. 示例结果

```
x = 0.5      GammaFunction(x) = 1.77245
x = 1        GammaFunction(x) = 1.00001
x = 1.5      GammaFunction(x) = 0.886227
x = 2        GammaFunction(x) = 1.00001
x = 2.5      GammaFunction(x) = 1.32934
x = 3        GammaFunction(x) = 2.00002
x = 3.5      GammaFunction(x) = 3.32335
x = 4        GammaFunction(x) = 6.00006
x = 4.5      GammaFunction(x) = 11.6317
x = 5        GammaFunction(x) = 24.0002
```

B(1.5,2.5) = 0.196348

16.2.2 不完全伽马函数

1. 功能

计算不完全伽马(Incomplete Camma)函数值。

2. 算法

不完全伽马函数的定义为

$$\Gamma(a,x)=\frac{P(a,x)}{\Gamma(a)},a>0,x>0 \tag{1}$$

其中

$$P(a,x)=\int_0^x \mathrm{e}^{-t}t^{a-1}\mathrm{d}t$$

对于不完全伽马函数有

$$\Gamma(a,0)=0,\Gamma(a,\infty)=1$$

不完全伽马函数也可表示为

$$\Gamma(a,x)=1-\frac{Q(a,x)}{\Gamma(a)},a>0,x>0 \tag{2}$$

其中

$$Q(a,x)=\int_0^{\infty} \mathrm{e}^{-t}t^{a-1}\mathrm{d}t$$

(2) 式中的 $Q(a,x)/\Gamma(a)$ 也称为余不完全伽马函数。

当 $x<a+1.0$ 时,用(1) 式计算。其中 $P(a,x)$ 用如下级数计算:

$$P(a,x)=\mathrm{e}^{-x}x^a\sum_{k=0}^{\infty}\frac{\Gamma(a)}{\Gamma(a+1+k)}x^k$$

当 $x\geqslant a+1$ 时,用(2) 式计算。其中 $Q(a,x)$ 用如下连分式计算:

$$Q(a,x)=\mathrm{e}^{-x}x^a\varphi(a,x)$$

$$\varphi(a,x)=\frac{1}{x+}\frac{1-a}{1+}\frac{1}{x+}\frac{2-a}{1+}\frac{2}{x+\cdots}+\frac{n-a}{1+}\frac{n}{x+\cdots}$$

3. 函数原型

```
//不完全伽马函数
template <class _Ty>
_Ty IncompleteGammaFunction(_Ty a, _Ty x);
```

4. 参数

a:_Ty 类型变量,自变量 a 的值。要求 $a>0$。当 $a\leqslant 0$ 时,本函数返回的函数为值 -1.0,且输出信息"Error * * $a<=0$!"。

x:_Ty 类型变量,自变量值。要求 $x>0$。当 $x<0$ 时,本函数返回的函数值为 -1.0,且输出信息"Error * * $x<0$!"。

5. 返回值

_Ty 类型。

当输入的参数值 $a \leqslant 0$ 时，本函数返回的函数为值 -1.0，且输出信息“Error * * $a <= 0$!”。

当输入的参数值 $x < 0$ 时，本函数返回函数值 -1.0，且输出信息“Error * * $x < 0$!”。

否则返回一个_Ty 类型的不完全伽马函数值。

6. 函数说明

(1) 本函数要调用计算伽马函数值的函数 GammaFunction()，参看 16.2.1 节。

(2) 本函数的输入、输出数据类型应为浮点数类型。

7. 示例

当 $a = 0.5, 5.0, 50.0$ 时，x 分别取 0.1, 1.0, 10.0 时的不完全伽马函数值 $\Gamma(a,x)$。

主函数及示例程序如下：

```
//IncompleteGammaFunction.cpp        不完全伽马函数

#include <iostream>                  //模板类输入、输出流标准头文件
#include <SpecialFunction.h>         //数学变换头文件
using namespace std;                 //名字空间

void main(void)
{
  double a[3] = {0.5,5.0,50.0};
    double x[3] = {0.1,1.0,10.0};

  for(int i = 0;i<3;i++)
  for(int j = 0;j<3;j++)
    {
      double s = a[i];
      double t = x[j];
      double y = IncompleteGammaFunction(s,t);
      cout << " GammaFunction(" << a[i] << "," << x[j]
                                          <<") \t = " << y << endl;
    }
    cout << endl;
  }
```

8. 示例结果

```
GammaFunction(0.5,0.1)        = 0.345279
GammaFunction(0.5,1)          = 0.842701
GammaFunction(0.5,10)         = 0.999992
GammaFunction(5,0.1)          = 7.66773e-008
GammaFunction(5,1)            = 0.00365981
GammaFunction(5,10)           = 0.970748
GammaFunction(50,0.1)         = 2.98087e-115
GammaFunction(50,1)           = 1.23374e-065
GammaFunction(50,10)          = 1.85471e-019
```

16.2.3 误差函数

1. 功能

计算实变量 x 的误差函数值

$$erf(x)=\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$$

2. 算法

误差函数 $erf(x)$具有下列极限值及对称性：

$$erf(0)=0,erf(\infty)=1,erf(-x)=-erf(x)$$

当 $x\geqslant 0$ 时，可以用不完全伽马函数计算误差函数，即

$$erf(x)=\Gamma(0.5,x^2)$$

利用误差函数可以计算余误差函数

$$erfc(x)=\frac{2}{\sqrt{2\pi}}\int_x^{\infty} e^{-t^2}dt=1-erf(x)$$

也可以计算正态概率积分

$$\varphi(x)=\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-\frac{t^2}{2}}dt=\frac{1}{2}+\frac{1}{2}erf\left(\frac{x}{\sqrt{2}}\right)$$

3. 函数原型

```
//误差函数
template <class _Ty>
_Ty ErrorFunction(_Ty x);
```

4. 参数

x：_Ty 类型变量。自变量值。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型误差函数值。

6. 函数说明

(1) 本函数要调用计算不完全伽马函数值的函数 IncompleteGammaFunction()，参看本章 16.2.2 节。

(2) 本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算 0～2 之间间隔为 0.05 的误差函数值。

主函数及示例程序如下：

```
//ErrorFunction.cpp                    误差函数

#include <iostream>                    //模板类输入、输出流标准头文件
#include <SpecialFunction.h>           //数学变换头文件
using namespace std;                   //名字空间

void main(void)
{
```

```
    double x, y;

    for(int i = 1; i<20;i ++ )
    {
        for(int j = 0;j<2;j ++ )
        {
            x = 0.05 * (2.0 * i + j);
            y = ErrorFunction(x);
            cout << "    ErrorFunction(" << x << ")\t = "<< y;
        }
            cout << endl;
        }

        x = 2.0;
        y = ErrorFunction(x);
        cout << "    ErrorFunction(" << x << ")\t = "<< y << endl;
    }
```

8. 示例结果

```
ErrorFunction(0.1) = 0.112463  ErrorFunction(0.15) = 0.167996
ErrorFunction(0.2) = 0.222703  ErrorFunction(0.25) = 0.276326
ErrorFunction(0.3) = 0.328627  ErrorFunction(0.35) = 0.379382
ErrorFunction(0.4) = 0.428392  ErrorFunction(0.45) = 0.475482
ErrorFunction(0.5) = 0.5205    ErrorFunction(0.55)  = 0.563323
ErrorFunction(0.6) = 0.603856  ErrorFunction(0.65) = 0.642029
ErrorFunction(0.7) = 0.677801  ErrorFunction(0.75) = 0.711156
ErrorFunction(0.8) = 0.742101  ErrorFunction(0.85) = 0.770668
ErrorFunction(0.9) = 0.796908  ErrorFunction(0.95) = 0.820891
ErrorFunction(1)   = 0.842701  ErrorFunction(1.05) = 0.862436
ErrorFunction(1.1) = 0.880205  ErrorFunction(1.15) = 0.896124
ErrorFunction(1.2) = 0.910314  ErrorFunction(1.25) = 0.9229
ErrorFunction(1.3) = 0.934008  ErrorFunction(1.35) = 0.943762
ErrorFunction(1.4) = 0.952285  ErrorFunction(1.45) = 0.959695
ErrorFunction(1.5) = 0.966105  ErrorFunction(1.55) = 0.971623
ErrorFunction(1.6) = 0.976348  ErrorFunction(1.65) = 0.980376
ErrorFunction(1.7) = 0.98379   ErrorFunction(1.75)  = 0.986672
ErrorFunction(1.8) = 0.989091  ErrorFunction(1.85) = 0.991111
ErrorFunction(1.9) = 0.99279   ErrorFunction(1.95) = 0.994179
ErrorFunction(2)   = 0.995322
```

16.2.4 第一类整数阶贝塞尔函数

1. 功能

计算实变量 x 的第一类整数阶贝塞耳(Bessel)函数值 $J_n(x)$。

2. 算法

第一类贝塞耳函数的定义为

$$J_n(x)=\left(\frac{x}{2}\right)^n\sum_{k=0}^{\infty}\frac{(-1)^k}{k!(n+k)!}\left(\frac{x}{2}\right)^{2k}$$

其中 n 为非负整数。其积分表达式为

$$J_n(x)=\frac{1}{2\pi}\int_{-\pi}^{\pi}\cos(nt-x\sin t)\,dt$$

第一类整数阶贝塞耳函数具有下列递推关系

$$J_{n+1}(x)=\frac{2n}{x}J_n(x)-J_{n-1}(x) \tag{1}$$

$J_0(x)$与 $J_1(x)$的计算公式如下：

当 $|x|<8.0$ 时，$J_0(x)=\dfrac{A(y)}{B(y)}$，其中 $y=x^2$，且

$$A(y)=a_0+a_1y+a_2y^2+a_3y^3+a_4y^4+a_5y^5$$
$$B(y)=b_0+b_1y+b_2y^2+b_3y^3+b_4y^4+b_5y^5$$

其系数分别为

$a_0=5.7568490574\times10^{10}$，　$a_1=-1.3362590354\times10^{10}$

$a_2=6.516196407\times10^{8}$，　$a_3=-1.121442418\times10^{7}$

$a_4=7.739233017\times10^{4}$，　$a_5=-1.849052456\times10^{2}$

$b_0=5.7568490411\times10^{10}$，　$b_1=1.029532985\times10^{9}$

$b_2=9.494680718\times10^{6}$，　$b_3=5.927264853\times10^{4}$

$b_4=2.678532712\times10^{2}$，　$b_5=1.0$

$$J_1(x)=x\frac{C(y)}{D(y)}$$

其中 $y=x\times x$，且

$$C(y)=c_0+c_1y+c_2y^2+c_3y^3+c_4y^4+c_5y^5$$
$$D(y)=d_0+d_1y+d_2y^2+d_3y^3+d_4y^4+d_5y^5$$

其系数分别为

$c_0=7.2362614232\times10^{10}$，　$c_1=-7.895059235\times10^{9}$

$c_2=2.423968531\times10^{8}$，　$c_3=-2.972611439\times10^{6}$

$c_4=1.570448260\times10^{4}$，　$c_5=-3.016036606\times10$

$d_0=1.44725228443\times10^{11}$，　$d_1=2.300535178\times10^{9}$

$d_2=1.858330474\times10^{7}$，　$d_3=9.944743994\times10^{4}$

$d_4=3.769991397\times10^{2}$，　$d_5=1.0$

当 $|x|\geqslant8.0$ 时，令 $z=8.0/|x|$，$y=z\times z$，

则

$$J_0(x)=\sqrt{\frac{2}{\pi|x|}}[E(y)\cos\theta-zF(y)\sin\theta]$$

其中 $\theta=|x|-\dfrac{\pi}{4}$，且

$$E(y)=e_0+e_1y+e_2y^2+e_3y^3+e_4y^4$$
$$F(y)=f_0+f_1y+f_2y^2+f_3y^3+f_4y^4$$

$e_0=1.0$，　$e_1=-0.1098628627\times10^{-2}$

$$e_2 = 0.2734510407 \times 10^{-4}, \quad e_3 = -0.2073370639 \times 10^{-5}$$
$$e_4 = 0.2093887211 \times 10^{-6}$$
$$f_0 = -0.1562499995 \times 10^{-1}, \quad f_1 = 0.1430488765 \times 10^{-3}$$
$$f_2 = -0.6911147651 \times 10^{-5}, \quad f_3 = 0.7621095161 \times 10^{-6}$$
$$f_4 = -0.934935152 \times 10^{-7}$$

而且

$$J_1(x) = \sqrt{\frac{2}{\pi |x|}}[G(y)\cos\theta - xH(y)\sin\theta], x > 0$$
$$J_1(-x) = -J_1(x)$$

其中 $\theta = |x| - \frac{3\pi}{4}$，且

$$G(y) = g_0 + g_1 y + g_2 y^2 + g_3 y^3 + g_4 y^4$$
$$H(y) = h_0 + h_1 y + h_2 y^2 + h_3 y^3 + h_4 y^4$$

其系数分别为

$$g_0 = 1.0, \quad g_1 = 0.183105 \times 10^{-2}$$
$$g_2 = -0.3516396496 \times 10^{-4}, \quad g_3 = 0.2457520174 \times 10^{-5}$$
$$g_4 = -0.240337019 \times 10^{-6}$$
$$h_0 = 0.4687499995 \times 10^{-1}, \quad h_1 = -0.2002690873 \times 10^{-3}$$
$$h_2 = 0.8449199096 \times 10^{-5}, \quad h_3 = -0.88228987 \times 10^{-6}$$
$$h_4 = 0.105787412 \times 10^{-6}$$

当 $n \geqslant 2$ 时，如果 $|x| > n$，则可用递推公式(1)进行递推计算；否则用下列递推关系

$$J_{n-1}(x) = \frac{2n}{x} J_n - J_{n+1}(x)$$

3. 函数原型

```
//第一类整数阶贝塞尔函数
template <class _Ty>
_Ty IntegerBessellstFunction(int n, _Ty x);
```

4. 参数

n：整型变量。第一类贝塞耳函数的阶数，$n \geqslant 0$ 。当 $n < 0$ 时，本函数按 $|n|$ 计算。

x：_Ty 类型变量。自变量值。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型的函数值 $J_n(x)$ 。

6. 函数说明

本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算 $n = 0,1,2,3,4,5$ 时，$x = 0.05, 0.5, 5.0, 50.0$ 时的 $J_n(x)$。

主函数及示例程序如下：

```
//IntegerBessellstFunction.cpp        第一类整数阶贝塞尔函数

#include <iostream>                   //模板类输入、输出流标准头文件
```

```
#include <SpecialFunction.h>        //数学变换头文件
using namespace std;                //名字空间

void main(void)
{
  double x, y;

  for(int n = 0; n<6; n++)
  {
      x = 0.05;
      for(int j = 1; j<5; j++)
      {
          y = IntegerBessel1stFunction(n, x);
          cout << "n = " << n << "  x = " << x
                  << "\t  IntegerBessel1stFunction(n, x) = " << y << endl;
          x = x * 10.0;
      }
    }
  }
```

8. 示例结果

```
n = 0  x = 0.05          IntegerBessel1stFunction(n, x) = 0.999375
n = 0  x = 0.5           IntegerBessel1stFunction(n, x) = 0.93847
n = 0  x = 5             IntegerBessel1stFunction(n, x) = -0.177597
n = 0  x = 50            IntegerBessel1stFunction(n, x) = 0.0558123
n = 1  x = 0.05          IntegerBessel1stFunction(n, x) = 0.0249922
n = 1  x = 0.5           IntegerBessel1stFunction(n, x) = 0.242268
n = 1  x = 5             IntegerBessel1stFunction(n, x) = -0.327579
n = 1  x = 50            IntegerBessel1stFunction(n, x) = -0.0975118
n = 2  x = 0.05          IntegerBessel1stFunction(n, x) = 0.000312435
n = 2  x = 0.5           IntegerBessel1stFunction(n, x) = 0.030604
n = 2  x = 5             IntegerBessel1stFunction(n, x) = 0.0465651
n = 2  x = 50            IntegerBessel1stFunction(n, x) = -0.0597128
n = 3  x = 0.05          IntegerBessel1stFunction(n, x) = 2.60376e-006
n = 3  x = 0.5           IntegerBessel1stFunction(n, x) = 0.00256373
n = 3  x = 5             IntegerBessel1stFunction(n, x) = 0.364831
n = 3  x = 50            IntegerBessel1stFunction(n, x) = 0.0927348
n = 4  x = 0.05          IntegerBessel1stFunction(n, x) = 1.6274e-008
n = 4  x = 0.5           IntegerBessel1stFunction(n, x) = 0.000160736
n = 4  x = 5             IntegerBessel1stFunction(n, x) = 0.391232
n = 4  x = 50            IntegerBessel1stFunction(n, x) = 0.070841
n = 5  x = 0.05          IntegerBessel1stFunction(n, x) = 8.13717e-011
n = 5  x = 0.5           IntegerBessel1stFunction(n, x) = 8.05363e-006
n = 5  x = 5             IntegerBessel1stFunction(n, x) = 0.261141
n = 5  x = 50            IntegerBessel1stFunction(n, x) = -0.0814002
```

16.2.5 第二类整数阶贝塞尔函数

1. 功能

计算实变量 x 的第二类整数阶贝塞耳(Bessel)函数值 $Y_n(x)$。

2. 算法

第二类整数阶贝塞耳函数具有如下递推关系

$$Y_{n+1}(x)=\frac{2n}{x}Y_n(x)-Y_{n-1}(x),\quad x\geqslant 0$$

这个递推公式是稳定的。

当 $x<8.0$ 时，$Y_0(x)=\dfrac{A(y)}{B(y)}+\dfrac{2}{\pi}J_0(x)\ln x$

其中 $y=x^2$，且

$$A(y)=a_0+a_1y+a_2y^2+a_3y^3+a_4y^4+a_5y^5$$
$$B(y)=b_0+b_1y+b_2y^2+b_3y^3+b_4y^4+b_5y^5$$

其系数分别为

$a_0=-2.957821389\times10^9$，　$a_1=7.062834065\times10^9$

$a_2=-5.123598036\times10^8$，　$a_3=7.087988129\times10^7$

$a_4=-8.632792757\times10^4$，　$a_5=2.284622733\times10^2$

$b_0=4.0076544269\times10^{10}$，　$b_1=7.452499648\times10^8$

$b_2=7.189466438\times10^6$，　$b_3=4.744726470\times10^4$

$b_4=2.261030244\times10^2$，　$b_5=1.0$

而

$$Y_1(x)=x\frac{C(y)}{D(y)}+\frac{2}{\pi}\left[J_1(x)\ln x-\frac{1}{x}\right]$$

其中 $y=x^2$，且

$$C(y)=c_0+c_1y+c_2y^2+c_3y^3+c_4y^4+c_5y^5$$
$$D(y)=d_0+d_1y+d_2y^2+d_3y^3+d_4y^4+d_5y^5+d_6y^6$$

其系数分别为

$c_0=-4.900604943\times10^{12}$，　$c_1=1.275274390\times10^{12}$

$c_2=-5.153438139\times10^{10}$，　$c_3=7.439264551\times10^8$

$c_4=-4.237922726\times10^6$，　$c_5=8.511937935\times10^3$

$d_0=2.499580570\times10^{13}$，　$d_1=4.244419664\times10^{11}$

$d_2=3.733650367\times10^9$，　$d_3=2.245904002\times10^7$

$d_4=1.020426050\times10^5$，　$d_5=3.549632885^2$

$d_6=1.0$

当 $x\geqslant 8.0$ 时，令 $z=8.0/x$，$y=z\times z$，

则　$Y_0(x)=\sqrt{\dfrac{2}{\pi x}}[E(y)\sin\theta+zF(y)\cos\theta]$

其中 $\theta=x-\dfrac{\pi}{4}$，且

$$E(y)=e_0+e_1y+e_2y^2+e_3y^3+e_4y^4$$

$$F(y) = f_0 + f_1 y + f_2 y^2 + f_3 y^3 + f_4 y^4$$

其系数分别为

$$e_0 = 1.0, \qquad e_1 = -0.1098628627 \times 10^{-2}$$
$$e_2 = 0.2734510407 \times 10^{-4}, \qquad e_3 = -0.2073370639 \times 10^{-5}$$
$$e_4 = 0.2093887211 \times 10^{-6}$$
$$f_0 = -0.1562499995 \times 10^{-1}, \qquad f_1 = 0.1430488765 \times 10^{-3}$$
$$f_2 = -0.6911147651 \times 10^{-5}, \qquad f_3 = 0.7621095161 \times 10^{-6}$$
$$f_4 = -0.934945152 \times 10^{-7}$$

而且

$$Y_1(x) = \sqrt{\frac{2}{\pi x}}[G(y)\sin\theta + zH(y)\cos\theta]$$

其中 $\theta = x - \frac{3\pi}{4}$，且

$$G(y) = g_0 + g_1 y + g_2 y^2 + g_3 y^3 + g_4 y^4$$
$$H(y) = h_0 + h_1 y + h_2 y^2 + h_3 y^3 + h_4 y^4$$

其系数分别为

$$g_0 = 1.0, \qquad g_1 = 0.183105 \times 10^{-2}$$
$$g_2 = -0.3516396496 \times 10^{-4}, \qquad g_3 = 0.2457520174 \times 10^{-5}$$
$$g_4 = -0.240337019 \times 10^{-6}$$
$$h_0 = 0.4687499995 \times 10^{-1}, \qquad h_1 = -0.2002690873 \times 10^{-3}$$
$$h_2 = 0.8449199096 \times 10^{-5}, \qquad h_3 = -0.88228987 \times 10^{-6}$$
$$h_4 = 0.105787412 \times 10^{-6}$$

3. 函数原型

```
//第二类整数阶贝塞尔函数
template <class _Ty>
_Ty IntegerBessel2ndFunction(int n, _Ty x);
```

4. 参数

n：整型变量。第二类贝塞耳函数的阶数，$n \geqslant 0$。当 $n < 0$ 时，本函数按 $|n|$ 计算。

x：_Ty 类型变量。自变量值，$x \geqslant 0$。当 $x < 0$ 时，本函数按 $|x|$ 计算。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型函数值 $Y_n(x)$。

6. 函数说明

(1) 本函数要调用第一类整数阶贝塞耳函数 IntegerBessel1stFunction()，参看本章 16.2.4 节。

(2) 本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算 $n = 0,1,2,3,4,5$ 时、$x = 0.05, 0.5, 5.0, 50.0$ 时的 $Y_n(x)$。

主函数及示例程序如下：

```
//IntegerBessel2ndFunction.cpp          第二类整数阶贝塞尔函数
```

```
#include <iostream>                    //模板类输入、输出流标准头文件
#include <SpecialFunction.h>           //数学变换头文件
using namespace std;                   //名字空间

void main(void)
{
  double x, y;

  for(int n = 0; n<6; n++)
  {
      x = 0.05;
      for(int j = 1; j<5; j++)
      {
          y = IntegerBessel2ndFunction(n, x);
          cout << "n = " << n << "  x = " << x
               << "\t  IntegerBessel2ndFunction(n, x) = " << y << endl;
          x = x * 10.0;
      }
  }
}
```

8. 示例结果

```
n = 0  x = 0.05      IntegerBessel2ndFunction(n, x) = -1.97931
n = 0  x = 0.5       IntegerBessel2ndFunction(n, x) = -0.444519
n = 0  x = 5         IntegerBessel2ndFunction(n, x) = -0.308518
n = 0  x = 50        IntegerBessel2ndFunction(n, x) = -0.098065
n = 1  x = 0.05      IntegerBessel2ndFunction(n, x) = -12.7899
n = 1  x = 0.5       IntegerBessel2ndFunction(n, x) = -1.47147
n = 1  x = 5         IntegerBessel2ndFunction(n, x) = 0.147863
n = 1  x = 50        IntegerBessel2ndFunction(n, x) = -0.0567957
n = 2  x = 0.05      IntegerBessel2ndFunction(n, x) = -509.615
n = 2  x = 0.5       IntegerBessel2ndFunction(n, x) = -5.44137
n = 2  x = 5         IntegerBessel2ndFunction(n, x) = 0.367663
n = 2  x = 50        IntegerBessel2ndFunction(n, x) = 0.0957932
n = 3  x = 0.05      IntegerBessel2ndFunction(n, x) = -40756.4
n = 3  x = 0.5       IntegerBessel2ndFunction(n, x) = -42.0595
n = 3  x = 5         IntegerBessel2ndFunction(n, x) = 0.146267
n = 3  x = 50        IntegerBessel2ndFunction(n, x) = 0.0644591
n = 4  x = 0.05      IntegerBessel2ndFunction(n, x) = -4.89026e+006
n = 4  x = 0.5       IntegerBessel2ndFunction(n, x) = -499.273
n = 4  x = 5         IntegerBessel2ndFunction(n, x) = -0.192142
n = 4  x = 50        IntegerBessel2ndFunction(n, x) = -0.0880581
n = 5  x = 0.05      IntegerBessel2ndFunction(n, x) = -7.82401e+008
n = 5  x = 0.5       IntegerBessel2ndFunction(n, x) = -7946.3
n = 5  x = 5         IntegerBessel2ndFunction(n, x) = -0.453695
```

```
n = 5  x = 50          IntegerBessel2ndFunction(n, x) = -0.0785484
```

16.2.6 变型第一类整数阶贝塞尔函数

1. 功能

计算实变量 x 的变型第一类整数阶贝塞耳(Bessel)函数值 $I_n(x)$。

2. 算法

第一类贝塞耳函数的定义为

$$I_n(x) = (-j)^n J_n(jx)$$

其中 n 为非负整数，j 为虚数(即$\sqrt{-1}$)，$J_n(jx)$为纯虚变量(jx)的第一类贝塞耳函数。

$I_0(x)$与 $I_1(x)$的计算公式如下

当$|x|<3.75$时，令 $y=\left(\frac{x}{3.75}\right)^2$，则

$$I_0(x) = a_0 + a_1 y + a_2 y^2 + a_3 y^3 + a_4 y^4 + a_5 y^5 + a_6 y^6$$
$$I_1(x) = x(b_0 + b_1 y + b_2 y^2 + b_3 y^3 + b_4 y^4 + b_5 y^5 + b_6 y^6)$$

其系数分别为

$a_0 = 1.0$，　　$a_1 = 3.5156229$

$a_2 = 3.0899424$，　　$a_3 = 1.2067492$

$a_4 = 2.659723 \times 10^{-1}$，　　$a_5 = 3.60768 \times 10^{-2}$

$a_6 = 4.5813 \times 10^{-3}$

$b_0 = 5.0 \times 10^{-1}$，　　$b_1 = 8.7890594 \times 10^{-1}$

$b_2 = 5.1498869 \times 10^{-1}$，　　$b_3 = 1.5084934 \times 10^{-1}$

$b_4 = 2.658773 \times 10^{-2}$，　　$b_5 = 3.01532 \times 10^{-3}$

$b_6 = 3.2411 \times 1.0^{-4}$

当$|x|\geqslant 3.75$时，令 $y=\frac{3.75}{|x|}$，则

$$I_0(x) = \frac{e^{|x|}}{\sqrt{|x|}} C(y)$$

其中

$$C(y) = c_0 + c_1 y + c_2 y^2 + c_3 y^3 + \cdots + c_8 y^8$$

且　　$I_1(|x|) = \frac{e^{|x|}}{\sqrt{|x|}} D(y)$，$I_1(-|x|) = -I_1(|x|)$

其中

$$D(y) = d_0 + d_1 y + d_2 y^2 + \cdots + d_8 y^8$$

其系数分别为

$c_0 = 3.9894228 \times 10^{-1}$，　　$c_1 = 1.328592 \times 10^{-2}$

$c_2 = 2.25319 \times 10^{-3}$，　　$c_3 = -1.57565 \times 10^{-3}$

$c_4 = 9.16281 \times 10^{-3}$，　　$c_5 = -2.057706 \times 10^{-2}$

$c_6 = 2.635537 \times 10^{-2}$，　　$c_7 = -1.647633 \times 10^{-2}$

$c_8 = 3.92377 \times 10^{-3}$

$d_0 = 3.9894228 \times 10^{-1}$，　　$d_1 = -3.988024 \times 10^{-2}$

$d_2 = -3.62018 \times 10^{-3}$，　　$d_3 = 1.63801 \times 10^{-3}$

$d_4 = -1.031555 \times 10^{-2}$，　　$d_5 = 2.282967 \times 10^{-2}$

$d_6 = -2.895312 \times 10^{-2}$，　　$d_7 = 1.787654 \times 10^{-2}$

$d_8 = -4.20059 \times 10^{-3}$

当 $n \geqslant 2$ 时，变型第一类贝塞耳函数具有下列递推关系

$$I_{n+1}(x) = -\frac{2n}{x} I_n(x) + I_{n-1}(x)$$

但这个递推公式是不稳定的。实际计算时，用如下递推关系

$$I_{n-1}(x) = \frac{2n}{x} I_n(x) + I_{n+1}(x)$$

3. 函数原型

```
//变型第一类整数阶贝塞尔函数
template <class _Ty>
_Ty TransformativeIntegerBessel1stFunction(int n,_Ty x);
```

4. 参数

n：整型变量。变型贝塞耳函数的阶数，$n \geqslant 0$ 。当 $n < 0$ 时，本函数按 $|n|$ 处理。

x：_Ty 类型变量。自变量值。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型函数值 $I_n(x)$。

6. 函数说明

本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算 $n = 0,1,2,3,4,5$、$x = 0.05,0.5,5.0,50.0$ 时的 $I_n(x)$。

主函数及示例程序如下：

```
//TransformativeIntegerBessel1stFunction.cpp  变型第一类整数阶贝塞尔函数

#include <iostream>              //模板类输入、输出流标准头文件
#include <SpecialFunction.h>     //数学变换头文件
using namespace std;             //名字空间

void main(void)
{
  double x, y;

  for(int n=0; n<6; n++)
  {
      x = 0.05;
      for(int j=1; j<5; j++)
      {
          y = TransformativeIntegerBessel1stFunction(n, x);
          cout << "n = " << n << "  x = " << x
```

```
                << "\t  TI(n, x) = " << y << endl;
            x = x * 10.0;
        }
    }
}
```

8. 示例结果

```
n = 0  x = 0.05          TI(n, x) = 1.00063
n = 0  x = 0.5           TI(n, x) = 1.06348
n = 0  x = 5             TI(n, x) = 27.2399
n = 0  x = 50            TI(n, x) = 2.93255e+020
n = 1  x = 0.05          TI(n, x) = 0.0250078
n = 1  x = 0.5           TI(n, x) = 0.257894
n = 1  x = 5             TI(n, x) = 24.3356
n = 1  x = 50            TI(n, x) = 2.90308e+020
n = 2  x = 0.05          TI(n, x) = 0.000312565
n = 2  x = 0.5           TI(n, x) = 0.0319061
n = 2  x = 5             TI(n, x) = 17.5056
n = 2  x = 50            TI(n, x) = 2.81647e+020
n = 3  x = 0.05          TI(n, x) = 2.60457e-006
n = 3  x = 0.5           TI(n, x) = 0.00264511
n = 3  x = 5             TI(n, x) = 10.3312
n = 3  x = 50            TI(n, x) = 2.67776e+020
n = 4  x = 0.05          TI(n, x) = 1.62781e-008
n = 4  x = 0.5           TI(n, x) = 0.000164806
n = 4  x = 5             TI(n, x) = 5.10823
n = 4  x = 50            TI(n, x) = 2.4951e+020
n = 5  x = 0.05          TI(n, x) = 8.13887e-011
n = 5  x = 0.5           TI(n, x) = 8.22317e-006
n = 5  x = 5             TI(n, x) = 2.15797
n = 5  x = 50            TI(n, x) = 2.27855e+020
```

16.2.7 变型第二类整数阶贝塞尔函数

1. 功能

计算实变量 x 的变型第二类整数阶贝塞耳(Bessel)函数值 $K_n(x)$。

2. 算法

变型第二类整数阶贝塞耳函数表示为

$$K_n(x) = \frac{\pi}{2} j^{n+1}[J_n(jx) + jY_n(jx)], \quad x > 0$$

其中 n 为非负整数，j 为虚数(即$\sqrt{-1}$)，$J_n(jx)$为纯虚变量(jx)的第一类贝塞耳函数，$Y_n(jx)$为纯虚变量(jx)的第二类贝塞耳函数。

$K_0(x)$与 $K_1(x)$的计算公式如下

当 $x \leqslant 2.0$ 时，令 $y = x^2/4.0$，则

$$K_0(x) = A(y) - I_0(x)\ln\left(\frac{x}{2}\right)$$

$$K_1(x) = \frac{1}{x}B(y) + I_1(x)\ln\left(\frac{x}{2}\right)$$

其中 $I_0(x)$、$I_1(x)$ 分别为变型第一类零阶、一阶贝塞耳函数，而

$$A(y) = a_0 + a_1 y + a_2 y^2 + a_3 y^3 + a_4 y^4 + a_5 y^5 + a_6 y^6$$

$$B(y) = b_0 + b_1 y + b_2 y^2 + b_3 y^3 + b_4 y^4 + b_5 y^5 + b_6 y^6$$

其系数为

$a_0 = -0.57721566$，　$a_1 = 0.42278420$

$a_2 = 0.23069759$，　$a_3 = 0.3488590 \times 10^{-1}$

$a_4 = 0.262698 \times 10^{-2}$，　$a_5 = 0.10750 \times 10^{-3}$

$a_6 = 0.74 \times 10^{-5}$

$b_0 = 1.0$，　$b_1 = 0.15443144$

$b_2 = -0.67278579$，　$b_3 = -0.18156897$

$b_4 = -0.1919402 \times 10^{-1}$，　$b_5 = -0.110404 \times 10^{-2}$

$b_6 = -0.4686 \times 1.0^{-4}$

当 $x > 2.0$ 时，令 $y = 2.0/x$，则

$$K_0(x) = \frac{e^{-x}}{\sqrt{x}}C(y)$$

$$K_1(x) = \frac{e^{-x}}{\sqrt{x}}D(y)$$

其中

$$C(y) = c_0 + c_1 y + c_2 y^2 + c_3 y^3 + c_4 y^4 + c_5 y^5 + c_6 y^6$$

$$D(y) = d_0 + d_1 y + d_2 y^2 + d_3 y^3 + d_4 y^4 + d_5 y^5 + d_6 y^6$$

其系数分别为

$c_0 = 0.125331414 \times 10^1$，　$c_1 = -0.7832358 \times 10^{-1}$

$c_2 = 0.2189568 \times 10^{-1}$，　$c_3 = -0.1062446 \times 10^{-1}$

$c_4 = 0.57872 \times 10^{-2}$，　$c_5 = -0.251540 \times 10^{-2}$

$c_6 = 0.53208 \times 10^{-3}$

$d_0 = 0.125331414 \times 10^1$，　$d_1 = 0.23498619$

$d_2 = -0.3655620 \times 10^{-1}$，　$d_3 = 0.1504268 \times 10^{-1}$

$d_4 = -0.780353 \times 10^{-2}$，　$d_5 = 0.325614 \times 10^{-2}$

$d_6 = -0.68245 \times 10^{-3}$

当 $n \geqslant 2$ 时，用递推关系式

$$K_{n+1}(x) = \frac{2n}{x}K_n(x) + K_{n-1}(x)$$

3. 函数原型

```
//变型第二类整数阶贝塞尔函数
template <class _Ty>
_Ty TransformativeIntegerBessel2ndFunction(int n, _Ty x);
```

4. 参数

n:整型变量。变型贝塞耳函数的阶数，$n \geqslant 0$。当 $n < 0$ 时，本函数按 $|n|$ 处理。

x:_Ty 类型变量。自变量值，$x > 0$ 。当 $x < 0$ 时，本函数按 $|x|$ 处理。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型函数值 $K_n(x)$。

6. 函数说明

(1) 本函数要调用变型第一类贝塞尔函数 TransformativeIntegerBessel1stFunction()，参看本章 16.2.6 节。

(2) 本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算 $n = 0,1,2,3,4,5$ 时，$x = 0.05,0.5,5.0,50.0$ 时的 $K_n(x)$。

主函数及示例程序如下：

```
//TransformativeIntegerBessel2ndFunction.cpp  变型第二类整数阶贝塞尔函数

#include <iostream>              //模板类输入、输出流标准头文件
#include <SpecialFunction.h>     //数学变换头文件
using namespace std;             //名字空间

void main(void)
{
  double x, y;

  for(int n=0; n<6; n++)
  {
      x = 0.05;
      for(int j=1; j<5; j++)
      {
          y = TransformativeIntegerBessel2ndFunction(n, x);
          cout << "n = " << n << "  x = " << x
               << "\t  TI(n, x) = " << y << endl;
          x = x * 10.0;
      }
  }
}
```

8. 示例结果

```
n = 0  x = 0.05        TI(n, x) = 3.11423
n = 0  x = 0.5         TI(n, x) = 0.924419
n = 0  x = 5           TI(n, x) = 0.0036911
n = 0  x = 50          TI(n, x) = 3.41017e-023
n = 1  x = 0.05        TI(n, x) = 19.9097
n = 1  x = 0.5         TI(n, x) = 1.65644
```

```
n = 1  x = 5          TI(n, x) = 0.00404461
n = 1  x = 50         TI(n, x) = 3.4441e-023
n = 2  x = 0.05       TI(n, x) = 799.501
n = 2  x = 0.5        TI(n, x) = 7.55018
n = 2  x = 5          TI(n, x) = 0.00530894
n = 2  x = 50         TI(n, x) = 3.54793e-023
n = 3  x = 0.05       TI(n, x) = 63980
n = 3  x = 0.5        TI(n, x) = 62.0579
n = 3  x = 5          TI(n, x) = 0.00829177
n = 3  x = 50         TI(n, x) = 3.72794e-023
n = 4  x = 0.05       TI(n, x) = 7.6784e+006
n = 4  x = 0.5        TI(n, x) = 752.245
n = 4  x = 5          TI(n, x) = 0.0152591
n = 4  x = 50         TI(n, x) = 3.99528e-023
n = 5  x = 0.05       TI(n, x) = 1.22861e+009
n = 5  x = 0.5        TI(n, x) = 12098
n = 5  x = 5          TI(n, x) = 0.0327063
n = 5  x = 50         TI(n, x) = 4.36718e-023
```

16.2.8 不完全贝塔函数

1. 功能

计算不完全贝塔(Beta)函数值 $B_x(a,b)$。

2. 算法

不完全贝塔函数的定义为

$$B_x(a,b)=\frac{1}{B(a,b)}\int_0^x t^{a-1}(1-t)^{b-1}\mathrm{d}t$$

其中 $a,b>0,0\leqslant x\leqslant 1,B(a,b)$ 为贝塔函数，即

$$B(a,b)=\frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

不完全贝塔函数具有下列对称关系及极限

$$B_x(a,b)=1-B_{1-x}(b,a) \tag{1}$$

$$B_0(a,b)=0,B_1(a,b)=1$$

不完全贝塔函数可以用下列连分式表示

$$B_x(a,b)=\frac{x^a(1-x)^b}{a\cdot B(a,b)}\varphi(x)$$

$$\varphi(x)=\frac{1}{1+\dfrac{d_1}{1+\dfrac{d_2}{1+\cdots}}} \tag{2}$$

其中

$$\left.\begin{aligned}d_{2k-1}&=-\frac{(a+k)(a+b+k)x}{(a+2k)(a+2k+1)}\\ d_{2k}&=\frac{k(b-k)x}{(a+2k-1)(a+2k)}\end{aligned}\right\}k=1,2,\cdots$$

当 $x<\frac{a+1}{a+b+2}$ 时，连分式(2)的收敛速度很快，而当 $x>\frac{a+1}{a+b+2}$ 时，可以用对称关系(1)来计算。

3. 函数原型

```
//不完全贝塔函数
template <class _Ty>
_Ty IncompleteBetaFunction(_Ty a, _Ty b, _Ty x);
```

4. 参数

a:_Ty 类型变量。不完全贝塔函数的参数，$a>0$。当 $a\leqslant 0$ 时，本函数输出信息“err * * $a<=0$!”，并返回函数值 -1。

b:_Ty 类型变量。不完全贝塔函数的参数，$b>0$。当 $b\leqslant 0$ 时，本函数输出信息“err * * $b<=0$!”，并返回函数值 -1。

x:_Ty 类型变量。不完全贝塔函数的自变量，$0\leqslant x\leqslant 1$。当 $x<0$ 或 $x>1$ 时，本函数输出信息“err * * $x<0$ or $x>1$!”，并返回值 10^{37}。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型函数值 $B_x(a,b)$。

6. 函数说明

(1) 本函数要调用伽马函数 GammaFunction()，参看本章 16.2.1 节。

(2) 本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算(a,b)为 (0.5,0.5)、(0.5,5.0)、(1.0,0.3)、(5.0,0.5)、(8.0,10.0)时，x 取值为 0.0、0.2、0.4、0.6、0.8、1.0 时的不完全贝塔函数值 $B_x(a,b)$。

主函数示例程序如下：

```
//IncompleteBetaFunction.cpp  不完全贝塔函数

#include <iostream>            //模板类输入、输出流标准头文件
#include <SpecialFunction.h>   //数学变换头文件
using namespace std;           //名字空间

void main(void)
{
  double x(0.0), y, a0, b0;
  double a[5] = { 0.5,0.5,1.0,5.0,8.0};
    double b[5] = { 0.5,5.0,3.0,0.5,10.0};

  for(int i=0; i<6; i++)
  {
      cout << "x = " << x << endl;
      for(int j=0; j<5; j++)
      {
          a0 = a[j];
```

```
            b0 = b[j];
            y = IncompleteBetaFunction(a0, b0, x);
            cout << "\tIncompleteBetaFunction(" << a0 << "," << b0
                                                 << ")\t = " << y << endl;
        }
        x = x + 0.2;
    }
}
```

8. 示例結果

```
x = 0
  IncompleteBetaFunction(0.5,0.5) = 0
  IncompleteBetaFunction(0.5,5)   = 0
  IncompleteBetaFunction(1,3)     = 0
  IncompleteBetaFunction(5,0.5)   = 0
  IncompleteBetaFunction(8,10)    = 0
x = 0.2
  IncompleteBetaFunction(0.5,0.5) = 0.273989
  IncompleteBetaFunction(0.5,5)   = 0.916541
  IncompleteBetaFunction(1,3)     = 0.36353
  IncompleteBetaFunction(5,0.5)   = 8.18696e-005
  IncompleteBetaFunction(8,10)    = 0.009585
x = 0.4
  IncompleteBetaFunction(0.5,0.5) = 0.365257
  IncompleteBetaFunction(0.5,5)   = 0.979039
  IncompleteBetaFunction(1,3)     = 0.856001
  IncompleteBetaFunction(5,0.5)   = 0.00271228
  IncompleteBetaFunction(8,10)    = 0.228213
x = 0.6
  IncompleteBetaFunction(0.5,0.5) = 0.634743
  IncompleteBetaFunction(0.5,5)   = 0.997288
  IncompleteBetaFunction(1,3)     = 0.947637
  IncompleteBetaFunction(5,0.5)   = 0.0209607
  IncompleteBetaFunction(8,10)    = 0.929174
x = 0.8
  IncompleteBetaFunction(0.5,0.5) = 0.726011
  IncompleteBetaFunction(0.5,5)   = 0.999918
  IncompleteBetaFunction(1,3)     = 0.992615
  IncompleteBetaFunction(5,0.5)   = 0.560966
  IncompleteBetaFunction(8,10)    = 0.999546
x = 1
  IncompleteBetaFunction(0.5,0.5) = 1
  IncompleteBetaFunction(0.5,5)   = 1
  IncompleteBetaFunction(1,3)     = 1
  IncompleteBetaFunction(5,0.5)   = 1
```

```
IncompleteBetaFunction(8,10)     = 1
```

16.2.9 正态分布函数

1. 功能

计算随机变量 x 的正态分布函数 $P(a,\sigma,x)$值。

2. 算法

正态分布函数的定义为

$$P(a,\sigma,x)=\frac{1}{\sqrt{2\pi}\sigma}\int_{-\infty}^{x}e^{-\frac{(t-a)^2}{2\sigma^2}}dt$$

其中 a 为随机变量的数学期望(平均值)；$\sigma>0$，σ^2 为随机变量的方差。

正态分布函数可以用误差函数来计算，即

$$P(a,\sigma,x)=\frac{1}{2}+\frac{1}{2}erf\left(\frac{x-a}{\sqrt{2}\sigma}\right)$$

3. 函数原型

```
//正态分布函数
template <class _Ty>
_Ty NormalDistributionFunction(_Ty a, _Ty d, _Ty x);
```

4. 参数

a：_Ty 类型变量。数学期望值。

b：_Ty 类型变量。d^2 为方差值，$d>0$ 。

x：_Ty 类型变量。随机变量值。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型函数值 $P(a,\sigma,x)$。

6. 函数说明

(1) 本函数要调用误差函数 ErrorFunction()，参看本章 16.2.3 节，误差函数又要调用不完全伽马函数 IncompleteGammaFunction()，参看本章 16.2.2 节。

(2) 本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算(a,σ)为(−1.0,0.5)、(3.0,15.0)时，x 取值为−10.0，−5.0，0.0，5.0 和 10.0 时的正态分布函数值 $P(a,\sigma,x)$。

主函数示例程序如下：

```
//NormalDistributionFunction.cpp  正态分布函数

#include <iostream>                 //模板类输入、输出流标准头文件
#include <SpecialFunction.h>        //数学变换头文件
using namespace std;                //名字空间

void main(void)
{
  double x, y, b, c;
```

```
    double a[2] = { -1.0,3.0};
      double d[2] = { 0.5,15.0};

    for(int i = 0; i<2; i++)
    {
        b = a[i];
        c = d[i];
        x = -10.0;
        for(int j = 0; j<5; j++)
        {
            y = NormalDistributionFunction(b, c, x);
            cout << " NormalDistributionFunction(" << b << "," << c
                                    << "," << x << ")\t= " << y << endl;
            x = x + 5.0;
        }

    }
}
```

8. 示例结果

```
NormalDistributionFunction(-1,0.5,-10)   = 0
NormalDistributionFunction(-1,0.5,-5)    = 6.10623e-016
NormalDistributionFunction(-1,0.5,0)     = 0.97725
NormalDistributionFunction(-1,0.5,5)     = 1
NormalDistributionFunction(-1,0.5,10)    = 1
NormalDistributionFunction(3,15,-10)     = 0.193062
NormalDistributionFunction(3,15,-5)      = 0.296901
NormalDistributionFunction(3,15,0)       = 0.42074
NormalDistributionFunction(3,15,5)       = 0.553035
NormalDistributionFunction(3,15,10)      = 0.679631
```

16.2.10 t-分布函数

1. 功能

计算 t-分布函数 $P(t,n)$。

2. 算法

t-分布又称为 Student-函数，定义为

$$P(t,n)=\frac{\Gamma\left(\frac{n+1}{2}\right)}{\sqrt{n\pi}\Gamma\left(\frac{n}{2}\right)}\int_{-t}^{t}\left(1+\frac{x^2}{n}\right)^{-\frac{n+1}{2}}\mathrm{d}x$$

其中 t 为随机变量，n 为自由度，且 $t\geqslant 0$。它的极限值为

$$P(0,n)=0,P(\infty,n)=1$$

t-分布函数可以用不完全贝塔函数表示，即

$$P(t,n)=1-\mathrm{B}_{\frac{n}{n+t^2}}\left(\frac{n}{2},0.5\right)$$

3. 函数原型

```
//t-分布函数
template <class _Ty>
_Ty tDistributionFunction(_Ty t, int n);
```

4. 参数

t:_Ty 类型变量。随机变量值，$t \geqslant 0$ 。

n:整型变量。自由度。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型函数值 P(t,n)。

6. 函数说明

(1) 本函数要调用不完全贝塔函数 IncompleteBetaFunction(),参看本章 16.2.8 节;不完全贝塔函数又要调有伽马函数 GammaFunction(),参看本章 16.2.1 节。

(2) 本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算 $n = 1,2,3$ 和 $4, t = 0.5$ 和 5.0 时 x^2 - 分布函数值 $P(x^2,n)$。

主函数示例程序如下：

```
//tDistributionFunction.cpp  t-分布函数

#include <iostream>              //模板类输入、输出流标准头文件
#include <SpecialFunction.h>     //数学变换头文件
using namespace std;             //名字空间

void main(void)
{
  double t, y;
  double a[2] = {-1.0, 3.0};
    double d[2] = { 0.5, 15.0};

  for(int i=1; i<6; i++)
  {
      t = 0.5;
      y = tDistributionFunction(t, i);
          cout << " tDistributionFunction(" << t << "," << i
                                          << ")\t= " << y << endl;
      t = 5.0;
      y = tDistributionFunction(t, i);
              cout << " tDistributionFunction(" << t << "," << i
                                              << ")\t= " << y << endl;

  }
}
```

8. 示例结果

```
tDistributionFunction(0.5,1)   = 0.273989
tDistributionFunction(5,1)     = 0.875927
tDistributionFunction(0.5,2)   = 0.311108
tDistributionFunction(5,2)     = 0.963219
tDistributionFunction(0.5,3)   = 0.325943
tDistributionFunction(5,3)     = 0.985174
tDistributionFunction(0.5,4)   = 0.333855
tDistributionFunction(5,4)     = 0.992855
tDistributionFunction(0.5,5)   = 0.338772
tDistributionFunction(5,5)     = 0.996117
```

16.2.11 χ^2-分布函数

1. 功能

计算 χ^2-分布函数值 $P(\chi^2,n)$。

2. 算法

χ^2-分布函数的定义为

$$P(\chi^2,n)=\frac{1}{2^{n/2}\Gamma\left(\frac{n}{2}\right)}\int_0^{\chi^2}t^{(n/2)-1}e^{-t/2}dt$$

其中 n 为自由度，且 $\chi^2\geqslant 0$。它的极限为

$$P(0,n)=0,P(\infty,n)=1$$

χ^2-分布函数可以用不完全伽马函数表示，即

$$P(\chi^2,n)=\Gamma\left(\frac{n}{2},\frac{\chi^2}{2}\right)$$

3. 函数原型

```
//X-2-分布函数
template <class _Ty>
_Ty X2DistributionFunction(_Ty x, int n);
```

4. 参数

t:_Ty 类型变量。χ^2 变量值。

n:整型变量。自由度。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型函数值 $P(\chi^2,n)$。

6. 函数说明

(1) 本函数要调用不完全伽马函数 IncompleteGammaFunction()，参看 16.2.2 节，不完全伽马函数又要调有伽马函数 GammaFunction()，参看本章 16.2.1 节。

(2) 本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算 $n=1,2,3,4$ 和 $5,t=0.5$ 和 5.0 时，χ^2- 分布函数值 $P(\chi^2,n)$。

主函数示例程序如下：

```
//X^2DistributionFunction.cpp X^2-分布函数

#include <iostream>                    //模板类输入、输出流标准头文件
#include <SpecialFunction.h>           //数学变换头文件
using namespace std;                   //名字空间

void main(void)
{
  double t, y;

  for(int i=1; i<6; i++)
  {
      t = 0.5;
      y = X2DistributionFunction(t, i);
          cout << " X2DistributionFunction(" << t << "," << i
                                             << ")\t= " << y << endl;
      t = 5.0;
      y = X2DistributionFunction(t, i);
           cout << " X2DistributionFunction(" << t << "," << i
                                             << ")\t= " << y << endl;

  }
}
```

8. 示例结果

```
X2DistributionFunction(0.5,1)     = 0.5205
X2DistributionFunction(5,1)       = 0.974653
X2DistributionFunction(0.5,2)     = 0.221197
X2DistributionFunction(5,2)       = 0.917916
X2DistributionFunction(0.5,3)     = 0.0811086
X2DistributionFunction(5,3)       = 0.828203
X2DistributionFunction(0.5,4)     = 0.0264988
X2DistributionFunction(5,4)       = 0.712695
X2DistributionFunction(0.5,5)     = 0.00787671
X2DistributionFunction(5,5)       = 0.58412
```

16.2.12 *F*-分布函数

1. 功能

计算随机变量 F 的 F-分布函数值 $P(F,n_1,n_2)$。

2. 算法

随机变量 F 的 F-分布函数的定义为

$$P(F,n_1,n_2)=\frac{\Gamma\left(\frac{n_1+n_2}{2}\right)}{\Gamma\left(\frac{n_1}{2}\right)\Gamma\left(\frac{n_2}{2}\right)}n_1^{n_1/2}n_2^{n_2/2}\int_F^{\infty}\frac{t^{n_1/2-1}}{(n_2+n_1t)^{(n_2+n_1)/2}}\mathrm{d}t$$

其中随机变量 $F \geqslant 0$，n_1 与 n_2 为自由度。它的极限为

$$P(0,n_1,n_2)=1, P(\infty,n_1,n_2)=0$$

F-分布函数可以用不完全贝塔函数计算，即

$$P(F,n_1,n_2)=B_{n_2/(n_2+n_1F)}\left(\frac{n_2}{2},\frac{n_1}{2}\right)$$

3. 函数原型

```
//F-分布函数
template <class _Ty>
_Ty FDistributionFunction(_Ty f, int n1, int n2);
```

4. 参数

t：_Ty 类型变量。随机变量 F 的取值，$F \geqslant 0$ 。

n_1：整型变量。自由度。

n_2：整型变量。自由度。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型函数值 $P(F,n_1,n_2)$。

6. 函数说明

(1) 本函数要调用不完全贝塔函数 IncompleteBetaFunction()，参看本章 16.2.8 节；不完全贝塔函数又要调有伽马函数 GammaFunction()，参看本章 16.2.1 节。

(2) 本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算(n_1，n_2)分别为(2,3)、(5,10)，随机变量 F 的取值为 3.50、9.0 时的 F-分布函数值 $P(F,n_1,n_2)$。

主函数示例程序如下：

```
//FDistributionFunction.cpp          F-分布函数

#include <iostream>                  //模板类输入、输出流标准头文件
#include <SpecialFunction.h>         //数学变换头文件
using namespace std;                 //名字空间

void main(void)
{
  int n1, n2;
  double f, y;
  int n[2] = {2,5};
    int m[2] = {3,10};

  for(int i = 0;i<2;i++)
  {
      n1 = n[i];
      n2 = m[i];
      f = 3.5;
```

```
        y = FDistributionFunction(f,n1,n2);
            cout << " FDistributionFunction(" << f << "," << n1
                                    << "," << n2 << ")\t = " << y << endl;
        f = 9.0;
        y = FDistributionFunction(f,n1,n2);
            cout << " FDistributionFunction(" << f << "," << n1
                                    << "," << n2 << ")\t = " << y << endl;

    }
}
```

8. 示例结果

```
FDistributionFunction(3.5,2,3)        = 0.138025
FDistributionFunction(9,2,3)          = 0.0502706
FDistributionFunction(3.5,5,10)       = 0.0357164
FDistributionFunction(9,5,10)         = 0.00168888
```

16.2.13 正弦积分

1. 功能

计算正弦积分函数值。

2. 算法

正弦积分函数的定义为

$$Si(x) = \int_0^x \frac{\sin t}{t} dt, x > 0$$

本函数采用勒让德-高斯求积公式计算该积分。

3. 函数原型

```
//正弦积分
template <class _Ty>
_Ty SineIntegralFunction(_Ty x);
```

4. 参数

x:_Ty 类型变量。自变量值 $x > 0$。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型正弦积分值。

6. 函数说明

(1) 本函数要调用不完全贝塔函数 IncompleteBetaFunction(),参看本章 16.2.8 节;不完全贝塔函数又要调有伽马函数 GammaFunction(),参看本章 16.2.1 节。

(2) 本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算自变量 x 从 0.5 开始,每隔 2.0 的 10 个正弦积分值。

主函数示例程序如下:

```
//SineIntegralFunction.cpp        正弦积分函数
```

```
#include <iostream>              //模板类输入、输出流标准头文件
#include <SpecialFunction.h>    //数学变换头文件
using namespace std;            //名字空间

void main(void)
{
  double x, y;

  for(int i = 0;i<10;i++)
  {
      x = 0.5 + i;
      y = SineIntegralFunction(x);
      cout << "  x = " << x << "   SineIntegralFunction(x) = "<< y << endl;
  }
}
```

8. 示例结果

```
x = 0.5    SineIntegralFunction(x) = 0.493107
x = 1.5    SineIntegralFunction(x) = 1.32468
x = 2.5    SineIntegralFunction(x) = 1.77852
x = 3.5    SineIntegralFunction(x) = 1.83313
x = 4.5    SineIntegralFunction(x) = 1.65414
x = 5.5    SineIntegralFunction(x) = 1.46872
x = 6.5    SineIntegralFunction(x) = 1.42179
x = 7.5    SineIntegralFunction(x) = 1.51068
x = 8.5    SineIntegralFunction(x) = 1.6296
x = 9.5    SineIntegralFunction(x) = 1.67446
```

16.2.14 余弦积分

1. 功能

计算余弦积分函数值。

2. 算法

余弦积分函数的定义为

$$Ci(x) = -\int_x^{\infty} \frac{\cos t}{t} dt, x > 0$$

计算余弦积分的公式为

$$Ci(x) = \gamma + \ln x - \int_0^x \frac{1-\cos t}{t} dt$$

其中

$$\gamma = 0.57721566490153286060651$$

为欧拉(Euler)常数。

本函数采用勒让德-高斯求积公式计算该积分

$$\int_0^x \frac{1-\cos t}{t} dt$$

3. 函数原型

```
//余弦积分
template <class _Ty>
_Ty CosineIntegralFunction(_Ty x);
```

4. 参数

x:_Ty 类型变量。自变量值。$x > 0$。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型余弦积分值。

6. 函数说明

本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算自变量 x 从 0.5 开始,每隔 2.0 的 10 个余弦积分值。

主函数示例程序如下:

```
//CosineIntegralFunction.cpp        余弦积分函数

#include <iostream>                 //模板类输入、输出流标准头文件
#include <SpecialFunction.h>        //数学变换头文件
using namespace std;                //名字空间

void main(void)
{
  double x, y;

  for(int i = 0; i<10; i++)
  {
      x = 0.5 + i;
      y = CosineIntegralFunction(x);
      cout << "  x = " << x << "  CosineIntegralFunction(x) = " << y << endl;
  }
}
```

8. 示例结果

```
x = 0.5    CosineIntegralFunction(x) = -0.177784
x = 1.5    CosineIntegralFunction(x) = 0.470356
x = 2.5    CosineIntegralFunction(x) = 0.285871
x = 3.5    CosineIntegralFunction(x) = -0.0321285
x = 4.5    CosineIntegralFunction(x) = -0.193491
x = 5.5    CosineIntegralFunction(x) = -0.142053
x = 6.5    CosineIntegralFunction(x) = 0.0111015
x = 7.5    CosineIntegralFunction(x) = 0.115633
x = 8.5    CosineIntegralFunction(x) = 0.0994314
x = 9.5    CosineIntegralFunction(x) = 0.00267806
```

16.2.15 指数积分

1. 功能

计算指数积分函数值。

2. 算法

指数积分函数的定义为

$$Ei(x) = -\int_x^{\infty} \frac{e^{-t}}{t} dt, x > 0$$

或

$$Ei(x) = -\int_{-\infty}^{x} \frac{e^{t}}{t} dt, x < 0$$

计算指数积分的公式为

$$Ei(x) = \gamma + \ln x + \int_0^x \frac{e^{-t} - 1}{t} dt, x > 0$$

其中

$$\gamma = 0.57721566490153286060651$$

为欧拉(Euler)常数。

本函数采用勒让德-高斯求积公式计算该积分

$$\int_0^x \frac{e^{-t} - 1}{t} dt$$

3. 函数原型

```
//指数积分
template <class _Ty>
_Ty ExponentIntegralFunction(_Ty x);
```

4. 参数

x:_Ty 类型变量。自变量值 $x > 0$ 。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型指数积分值。

6. 函数说明

本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算自变量 x 从 0.05 开始,每隔 0.2 的 10 个余弦积分值。

主函数示例程序如下:

```
//ExponentIntegralFunction.cpp        指数积分函数

#include <iostream>                   //模板类输入、输出流标准头文件
#include <SpecialFunction.h>          //数学变换头文件
using namespace std;                  //名字空间

void main(void)
{
```

```
    double x, y;

    for(int i = 0; i<10; i++)
    {
        x = 0.05 + 0.2 * i;
        y = ExponentIntegralFunction(x);
        cout << "  x = " << x << "   ExponentIntegralFunction(x) = " << y << endl;
    }
}
```

8. 示例结果

```
x = 0.05    ExponentIntegralFunction(x) = -2.4679
x = 0.25    ExponentIntegralFunction(x) = -1.04428
x = 0.45    ExponentIntegralFunction(x) = -0.625331
x = 0.65    ExponentIntegralFunction(x) = -0.411517
x = 0.85    ExponentIntegralFunction(x) = -0.284019
x = 1.05    ExponentIntegralFunction(x) = -0.201873
x = 1.25    ExponentIntegralFunction(x) = -0.146413
x = 1.45    ExponentIntegralFunction(x) = -0.107777
x = 1.65    ExponentIntegralFunction(x) = -0.0802476
x = 1.85    ExponentIntegralFunction(x) = -0.060295
```

16.2.16 第一类椭圆积分

1. 功能

计算第一类椭圆积分。

2. 算法

第一类椭圆积分的定义为

$$F(k,\varphi)=\int_0^{\varphi}\frac{1}{\sqrt{1-k^2\sin^2\theta}}\mathrm{d}\theta,0\leqslant k\leqslant 1$$

当 $\varphi=\frac{\pi}{2}$ 时，$F\left(k,\frac{\pi}{2}\right)$ 称为第一类完全椭圆积分。

当 $|\varphi|>\frac{\pi}{2}$ 时，第一类椭圆积分有如下关系

$$F(k,n\pi\pm\varphi)=2\mathrm{n}F\left(k,\frac{\pi}{2}\right)\pm F(k,\varphi)$$

本函数采用勒让德—高斯求积公式计算上述积分。

3. 函数原型

```
//第一类椭圆积分
template <class _Ty>
_Ty EllipselstIntegral(_Ty k, _Ty f);
```

4. 参数

k：_Ty 类型变量。要求 $0\leqslant k\leqslant 1$。

f：_Ty 类型变量。椭圆积分中的 φ。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型的第一类椭圆积分值。

6. 函数说明

本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算自变量 $k = 0.5$ 与 1.0，φ 取 $\varphi_i = \frac{\pi}{18}i(i = 0,1,\cdots,10)$ 时，第一类椭圆积分的值。

主函数示例程序如下：

```
//EllipseIstIntegral.cpp              第一类椭圆积分

#include <iostream>                   //模板类输入、输出流标准头文件
#include <SpecialFunction.h>          //数学变换头文件
using namespace std;                  //名字空间

void main(void)
{
  double f, k, y;

  for(int i = 0; i<11; i++)
  {
      f = i * 3.1415926 / 18.0;
      k = 0.5;
      y = EllipseIstIntegral(k,f);
          cout << " EllipseIstIntegral(" << k << "," << f
                                         << ")\t   = " << y << endl;
      k = 1.0;
      y = EllipseIstIntegral(k,f);
          cout << " EllipseIstIntegral(" << k << "," << f
                                         << ")\t = " << y << endl;
  }
}
```

8. 示例结果

```
EllipseIstIntegral(0.5,0)         = 0
EllipseIstIntegral(1,0)           = 0
EllipseIstIntegral(0.5,0.174533)= 0.174754
EllipseIstIntegral(1,0.174533)    = 0.175426
EllipseIstIntegral(0.5,0.349066)= 0.350819
EllipseIstIntegral(1,0.349066)    = 0.356378
EllipseIstIntegral(0.5,0.523599)= 0.529429
EllipseIstIntegral(1,0.523599)    = 0.549306
EllipseIstIntegral(0.5,0.698132)= 0.711647
EllipseIstIntegral(1,0.698132)    = 0.76291
```

```
Ellipse1stIntegral(0.5,0.872665)= 0.898245
Ellipse1stIntegral(1,0.872665)  = 1.01068
Ellipse1stIntegral(0.5,1.0472)  = 1.08955
Ellipse1stIntegral(1,1.0472)    = 1.31696
Ellipse1stIntegral(0.5,1.22173) = 1.2853
Ellipse1stIntegral(1,1.22173)   = 1.73542
Ellipse1stIntegral(0.5,1.39626) = 1.48455
Ellipse1stIntegral(1,1.39626)   = 2.43625
Ellipse1stIntegral(0.5,1.5708)  = 1.68575
Ellipse1stIntegral(1,1.5708)    = 13.8106
Ellipse1stIntegral(0.5,1.74533) = 1.88695
Ellipse1stIntegral(1,1.74533)   = 25.185
```

16.2.17 第二类椭圆积分

1. 功能

计算第二类椭圆积分。

2. 算法

第二类椭圆积分的定义为

$$E(k,\varphi)=\int_0^{\varphi}\sqrt{1-k^2\sin^2\theta}\mathrm{d}\theta,0\leqslant k\leqslant 1$$

当 $\varphi=\frac{\pi}{2}$ 时，$E\left(k,\frac{\pi}{2}\right)$ 称为第二类完全椭圆积分。

当 $|\varphi|>\frac{\pi}{2}$ 时，第二类椭圆积分有如下关系

$$E(k,n\pi\pm\varphi)=2nE\left(k,\frac{\pi}{2}\right)\pm E(k,\varphi)$$

本函数采用勒让德—高斯求积公式计算上述积分。

3. 函数原型

```
//第二类椭圆积分
template <class _Ty>
_Ty Ellipse2ndIntegral(_Ty k, _Ty f);
```

4. 参数

k：_Ty 类型变量。要求 $0\leqslant k\leqslant 1$。

f：_Ty 类型变量。椭圆积分中的 φ。

5. 返回值

_Ty 类型。

本函数返回一个_Ty 类型的第二类椭圆积分值。

6. 函数说明

本函数的输入、输出的数据类型应为浮点数类型。

7. 示例

计算自变量 $k=0.5$ 与 1.0，φ 取 $\varphi_i=\frac{\pi}{18}i(i=0,1,\cdots,10)$，计算第二类椭圆积分值。

主函数示例程序如下：

```
/ //Ellipse2ndIntegral. cpp          第二类椭圆积分

# include <iostream>                 //模板类输入、输出流标准头文件
# include <SpecialFunction. h>       //数学变换头文件
using namespace std;                 //名字空间

void main(void)
{
  double f, k, y;
  for(int i = 0; i<11; i++)
  {
      f = i * 3.1415926 / 18.0;
      k = 0.5;
      y = Ellipse2ndIntegral(k, f);
          cout << "Ellipse2ndIntegral(" << k << "," << f
                                        << ")\t   = " << y << endl;
      k = 1.0;
      y = Ellipse2ndIntegral(k, f);
          cout << "Ellipse2ndIntegral(" << k << "," << f
                                        << ")\t   = " << y << endl;
  }
}
```

8. 示例结果

```
Ellipse2ndIntegral(0.5,0)         = 0
Ellipse2ndIntegral(1,0)           = 0
Ellipse2ndIntegral(0.5,0.174533)= 0.174312
Ellipse2ndIntegral(1,0.174533)    = 0.173648
Ellipse2ndIntegral(0.5,0.349066)= 0.347329
Ellipse2ndIntegral(1,0.349066)    = 0.34202
Ellipse2ndIntegral(0.5,0.523599)= 0.517882
Ellipse2ndIntegral(1,0.523599)    = 0.5
Ellipse2ndIntegral(0.5,0.698132)= 0.68506
Ellipse2ndIntegral(1,0.698132)    = 0.642788
Ellipse2ndIntegral(0.5,0.872665)= 0.848317
Ellipse2ndIntegral(1,0.872665)    = 0.766044
Ellipse2ndIntegral(0.5,1.0472)    = 1.00756
Ellipse2ndIntegral(1,1.0472)      = 0.866025
Ellipse2ndIntegral(0.5,1.22173)   = 1.16318
Ellipse2ndIntegral(1,1.22173)     = 0.939693
Ellipse2ndIntegral(0.5,1.39626)   = 1.31606
Ellipse2ndIntegral(1,1.39626)     = 0.984808
Ellipse2ndIntegral(0.5,1.5708)    = 1.46746
```

```
Ellipse2ndIntegral(1,1.5708)     = 1
Ellipse2ndIntegral(0.5,1.74533) = 1.61887
Ellipse2ndIntegral(1,1.74533)    = 1.01519
```

参考文献

1. 官章全. 标准C++库大全. 北京:电子工业出版社,2002

2. [美]Al Stevens, Clayton Walnum 著,林丽闽,别红霞等译. 标准C++宝典. 北京:电子工业出版社,2001

3. MSDN Library Help. Microsoft Corp. ,2000

4. [美]Brian Overland 著,董梁,李君成,李自更等译. C++语言命令详解(第二版). 北京:电子工业出版社,2000

5. [美]Bruce Eckel 著,刘宗田,邢大红,孙慧杰等译. C++编程思想 Thinking in C++. 北京:机械工业出版社,2000

6. [美]Jesse Liberty 著,郭强,宁黎松,韩珊译. C++编程实例详解. 北京:电子工业出版社,1999

7. Golub G. 等著,廉庆荣等译,熊西文审校. 矩阵计算. 大连:大连理工大学出版社,1998

8. Dahlquist G. 等著,包雪松译,唐述钊校. 数值方法. 北京:北京科学出版社,1990

9. Forsythe G. 等著,计九三译,孙增圻校. 计算机数值计算方法. 北京:清华大学出版社,1987

10. 威尔金森 J. H. 著,石钟慈,邓健新译. 代数特征值问题. 北京:北京科学出版社,1987

11. 奥特加 J. M. ,莱因博尔 W. C. 著,朱季纳译,孙念增,进友校. 多元非线性方程组迭代解法. 北京:北京科学出版社,1983

12. 李庆扬,王能超,易大义. 数值分析. 武汉:华中理工学院出版社,1982

13. 王德人. 非线性方程组解法与最优化方法. 北京:人民教育出版社,1979

14. 数学手册. 北京:人民教育出版社,1979

15. 吉尔 C. W. 著,费景高等译. 常微分方程组初值问题数值解法. 北京:北京科学出版社,1978

16. 徐士良主编. C 常用算法程序集. 北京:清华大学出版社,1996

17. 向光渝. Visual C++常用数值算法集. 北京:科学出版社,2002

参考文献

1. 官章全. 标准C++库大全. 北京:电子工业出版社,2002

2. [美]Al Stevens, Clayton Walnum 著,林丽闽,别红霞等译. 标准C++宝典. 北京:电子工业出版社,2001

3. MSDN Library Help. Microsoft Corp. ,2000

4. [美]Brian Overland 著,董梁,李君成,李自更等译. C++语言命令详解(第二版). 北京:电子工业出版社,2000

5. [美]Bruce Eckel 著,刘宗田,邢大红,孙慧杰等译. C++编程思想 Thinking in C++. 北京:机械工业出版社,2000

6. [美]Jesse Liberty 著,郭强,宁黎松,韩珊译. C++编程实例详解. 北京:电子工业出版社,1999

7. Golub G. 等著,廉庆荣等译,熊西文审校. 矩阵计算. 大连:大连理工大学出版社,1998

8. Dahlquist G. 等著,包雪松译,唐述钊校. 数值方法. 北京:北京科学出版社,1990

9. Forsythe G. 等著,计九三译,孙增圻校. 计算机数值计算方法. 北京:清华大学出版社,1987

10. 威尔金森 J. H. 著,石钟慈,邓健新译. 代数特征值问题. 北京:北京科学出版社,1987

11. 奥特加 J. M. ,莱因博尔 W. C. 著,朱季纳译,孙念增,进友校. 多元非线性方程组迭代解法. 北京:北京科学出版社,1983

12. 李庆扬,王能超,易大义. 数值分析. 武汉:华中理工学院出版社,1982

13. 王德人. 非线性方程组解法与最优化方法. 北京:人民教育出版社,1979

14. 数学手册. 北京:人民教育出版社,1979

15. 吉尔 C. W. 著,费景高等译. 常微分方程组初值问题数值解法. 北京:北京科学出版社,1978

16. 徐士良主编. C 常用算法程序集. 北京:清华大学出版社,1996

17. 向光渝. Visual C++常用数值算法集. 北京:科学出版社,2002

参考文献

1. 官章全. 标准C++库大全. 北京:电子工业出版社,2002

2. [美]Al Stevens, Clayton Walnum 著,林丽闽,别红霞等译. 标准C++宝典. 北京:电子工业出版社,2001

3. MSDN Library Help. Microsoft Corp. ,2000

4. [美]Brian Overland 著,董梁,李君成,李自更等译. C++语言命令详解(第二版). 北京:电子工业出版社,2000

5. [美]Bruce Eckel 著,刘宗田,邢大红,孙慧杰等译. C++编程思想 Thinking in C++. 北京:机械工业出版社,2000

6. [美]Jesse Liberty 著,郭强,宁黎松,韩珊译. C++编程实例详解. 北京:电子工业出版社,1999

7. Golub G. 等著,廉庆荣等译,熊西文审校. 矩阵计算. 大连:大连理工大学出版社,1998

8. Dahlquist G. 等著,包雪松译,唐述钊校. 数值方法. 北京:北京科学出版社,1990

9. Forsythe G. 等著,计九三译,孙增圻校. 计算机数值计算方法. 北京:清华大学出版社,1987

10. 威尔金森 J. H. 著,石钟慈,邓健新译. 代数特征值问题. 北京:北京科学出版社,1987

11. 奥特加 J. M. ,莱因博尔 W. C. 著,朱季纳译,孙念增,进友校. 多元非线性方程组迭代解法. 北京:北京科学出版社,1983

12. 李庆扬,王能超,易大义. 数值分析. 武汉:华中理工学院出版社,1982

13. 王德人. 非线性方程组解法与最优化方法. 北京:人民教育出版社,1979

14. 数学手册. 北京:人民教育出版社,1979

15. 吉尔 C. W. 著,费景高等译. 常微分方程组初值问题数值解法. 北京:北京科学出版社,1978

16. 徐士良主编. C 常用算法程序集. 北京:清华大学出版社,1996

17. 向光渝. Visual C++常用数值算法集. 北京:科学出版社,2002

计算机
常用数值算法与程序
（C++版）



ISBN 7-115-11183-9
9 787115 111838 >

ISBN7-115-11183-9/TP·3393
定价：49.00 元（附光盘）